Universidade Federal de Pernambuco
Graduação em Ciência da Computação

Centro de Informática
2009.1

IMPLEMENTAÇÃO DO RANSAC COM ALGORITMO DE *EIGHT-POINT* EM GPU

Trabalho de Graduação

ANDRÉ VITOR DE ALMEIDA PALHARES

Orientador: Prof. Djamel Sadok (jamel@cin.ufpe.br)

Recife,
2009

Universidade Federal de Pernambuco
Graduação em Ciência da Computação

Centro de Informática
2009.1

IMPLEMENTAÇÃO DO RANSAC COM ALGORITMO DE *EIGHT-POINT* EM GPU

Trabalho de Graduação

ANDRÉ VITOR DE ALMEIDA PALHARES

Projeto de Graduação apresentado no Centro de Informática da Universidade Federal de Pernambuco por André Vitor de Almeida Palhares, orientado pelo PhD. Djamel Sadok, como requisito parcial para a obtenção do grau de Engenheiro da Computação

Orientador: Prof. Djamel Sadok (jamel@cin.ufpe.br)

Recife,
2009

FOLHA DE APROVAÇÃO

IMPLEMENTAÇÃO DO RANSAC COM ALGORITMO DE *EIGHT-POINT* EM GPU

ANDRÉ VITOR DE ALMEIDA PALHARES

APROVADO EM 29 DE JUNHO DE 2009

BANCA EXAMINADORA:

_____

Prof. Djamel Fawzi Hadj Sadok, PhD –
UFPE (Orientador)

_____

Prof. Veronica Teichrieb, PhD –
UFPE (Avaliadora)

“Se fui capaz de ver mais longe foi apenas porque eu estava apoiado sobre ombro de gigantes.”

Albert Einstein

A minha família e amigos

Agradecimentos

Inicialmente, gostaria de agradecer às pessoas com quem convivo desde que nasci e que sem dúvidas contribuíram para que esse trabalho pudesse um dia ser feito, seja com os ensinamentos que me passaram ou através da autoridade deles que me fez seguir o caminho certo. São eles meu pai e minha mãe, Sylvio e Eduvirges. Também agradeço a minha irmã Sylvia, apesar das pequenas brigas. Mas não existe relacionamento sem brigas. Também gostaria de agradecer especialmente a Camila, que surgiu há pouco tempo na minha vida, mas que já me trouxe muitas felicidades e apoio.

Aqui também quero lembrar aqueles que durante o curso sempre estavam lá para ajudar e discutir problemas e que, cada um da sua maneira, ajudaram a moldar o meu caráter e personalidade como eles são hoje. São eles Thiago Arruda, Icamaan, Christian, Arthur Gonçalves e Renato, que apesar de não serem exatamente do meu curso, nunca deixaram de conviver comigo (apenas agora que estão fazendo mestrado em outros lugares). Também aqui tem um espaço reservado para o pessoal que começou o curso comigo, alguns deles finalizando essa jornada de longos anos. Estes são Jesus, Josias, Marcelo e, em especial, Rafael.

Também agradeço ao pessoal do GPRT, Glauco, Ramide e agora Rodolpho e Rodrigo, que, apesar das dificuldades, sempre conseguem resolver os problemas mais complicados com o menor número de ferramentas possível.

Por fim, também quero agradecer ao professor Djamel Sadok por ter me apoiado no GPRT e ter permitido que eu fizesse este trabalho numa área com a qual ele não possui grande afinidade. Apesar disto, creio que assimilar conhecimentos é sempre bom e Djamel não é contrário a essa idéia. Também agradeço ao professor Sílvio Melo, com quem discuto sobre alguns assuntos interessantes quando encontro. Não poderia esquecer da professora Veronica Teichrieb, por ter dado a idéia inicial deste trabalho. Também, a professora Judith Kelner, com quem convivo há quase dois anos e que sempre demonstrou boa vontade para resolver qualquer tipo de problema.

Resumo

O algoritmo de *eight-point* normalizado é um importante algoritmo utilizado na obtenção inicial de uma matriz fundamental de uma cena capturada a partir de duas câmeras descalibradas, visando, por exemplo, a reconstrução 3D desta cena. Na obtenção de uma matriz fundamental com maior precisão, o *eight-point* pode ser usado em conjunto com métodos iterativos como o RANSAC (RANdom SAmple Consensus), um estimador robusto, com pouca suscetibilidade a ruídos nos dados. Uma grande melhoria no desempenho destes algoritmos pode ser obtida numa implementação em paralelo utilizando as modernas GPUs de propósito geral. É neste contexto que se apresenta este projeto, que visa obter uma implementação eficiente em GPU e compará-la a implementações padrão desses algoritmos.

Sumário

Índice de Figuras

Índice de Tabelas

1 Introdução

Este capítulo visa introduzir o leitor no contexto em que o trabalho está inserido, apresentando uma motivação para o seu desenvolvimento. O problema em si será então caracterizado e os objetivos do trabalho serão elucidados.

1.1 Contexto

O presente trabalho se encaixa no contexto de reconstrução 3D, utilizando-se do modelo de programação em paralelo de CUDA. As duas seções abaixo explicam melhor esses conceitos iniciais.

1.1.1 Reconstrução 3D

Este trabalho está inserido no contexto de reconstrução 3D a partir de cenas capturadas a partir de diferentes visões (câmeras). Quando falamos de reconstrução 3D, falamos da criação de modelos espaciais que visam dar uma noção mais realística de uma determinada cena que foi obtida a partir de imagens.

Vale a pena citar a importância da reconstrução 3D, para a qual existem diversas áreas de aplicações. Entre elas, podemos citar as de documentação arquitetônica e arqueológica, nas quais é de fundamental importância, pois um modelo 3D pode ser utilizado, por exemplo, como base para propósitos de conservação ou restauração do ambiente; engenharia, arquitetura, planejamento urbano e turismo, nas quais modelos podem ser exibidos de modo a impressionar o observador ou para análise do ponto de vista de viabilidade de obras; também há aplicações na área médica, de reconstrução de órgãos humanos para um estudo mais profundo sobre anatomia ou no diagnóstico de doenças; na área cinematográfica para exibição de cenas com alto grau de realidade e fidelidade, sem a necessidade de construção manual completa da cena. Alguns exemplos claros de uso de técnicas de reconstrução 3D em algumas das áreas citadas podem ser encontrados em [20], [21], [22] e [23]. Claro que a reconstrução 3D não se resume a apenas estas aplicações.

Existem diversas técnicas de reconstrução 3D. Uma das principais (e de maior importância para esse trabalho) utiliza como entrada um conjunto de imagens obtidas a partir de diferentes ângulos de um determinado objeto ou ambiente. Entre estas, podemos citar a

Structure from Motion (SfM) [17], na qual uma cena estática é filmada, obtendo-se diversas imagens da cena e, a partir delas, é feita a reconstrução.

1.1.2 *Graphics Processing Units* e CUDA

A GPU (*Graphics Processing Unit*) [16] é um tipo de microprocessador especializado no processamento de gráficos. Este dispositivo é encontrado em *videogames*, computadores pessoais e estações de trabalho, e pode estar situado numa placa de vídeo ou integrado diretamente à placa mãe.

As GPUs modernas são responsáveis pela renderização e processamento de gráficos de um computador. A sua estrutura extremamente paralela permite que diversas operações sejam realizadas concorrentemente, realizando assim tarefas complexas com grande eficiência, as quais seriam extremamente custosas para uma CPU (*Central Processing Unit*) comum.

Com o advento de GPGPUs (*General-Purpose computing on Graphics Processing Units*, ou computação de propósito geral em GPUs) [15], não apenas operações de origem estritamente gráfica são realizadas em GPUs. GPGPUs são capazes de realizar diversas tarefas dos mais variados tipos de aplicações que possam ser solucionadas através de algoritmos paralelizáveis com extrema eficiência.

Um importante desafio no contexto de GPGPUs é o de como ocorrerá o desenvolvimento de *softwares* voltados para esse novo paradigma de processamento. Surge assim a necessidade de criar um ambiente no qual o programador possa acessar os mais diversos tipos de recursos desses novos tipos de processadores, bem como criar aplicações paralelas de um modo transparente e escalável, independente do número de processadores.

CUDA (*Compute Unified Device Architecture*) [13], [24] é um ambiente de *software* com um modelo de programação paralela que foi projetado para lidar com este problema. É baseado na linguagem C, facilitando assim o aprendizado por parte de programadores que têm conhecimentos nessa linguagem.

No ambiente CUDA existem alguns conceitos chaves, entre eles uma hierarquia de grupos de *threads* e de memória. Estes conceitos são disponibilizados ao programador a partir de um conjunto de extensões à linguagem C, e servem para gerenciar os aspectos de paralelização da plataforma.

1.2 Caracterização do Problema

A reconstrução 3D através da SfM consiste em algumas etapas [17]. Resumindo-as, temos o seguinte:

1. Aquisição de imagens: aqui, coletamos diferentes imagens de uma cena utilizando uma câmera de vídeo ou uma seqüência de fotografias.
2. Seleção de *features*: as *features* são pontos de alto contraste e destaque nas imagens e, portanto são facilmente identificadas.
3. Correspondência e rastreamento de *features*: as *features* identificadas são colocadas em correspondência entre diferentes imagens da cena.
4. Reconstrução projetiva: recuperação da movimentação da câmera e da posição espacial das *features*.
5. Reconstrução métrica: obtenção de uma transformação que permite identificar as posições 3D das *features* dado um sistema métrico.
6. Reconstrução densa: uso dos parâmetros da câmera, adquiridos em etapas anteriores para recuperação de todos os pontos das imagens.
7. Geração da malha e texturização: uso das cores originais da imagem de modo a dar uma aparência visual semelhante ao modelo tridimensional.

Durante o processo de reconstrução, existe a necessidade de se descobrir a matriz fundamental, que é uma importante matriz que caracteriza a estrutura tridimensional do ambiente e será utilizada durante algumas dessas etapas. Existem diversos algoritmos que visam recuperar a matriz fundamental, entre eles o algoritmo de *eight-point* [2].

O *eight-point* utiliza no mínimo oito correspondências entre pontos nas imagens para obter a matriz fundamental (daí seu nome). Porém, antes da recuperação dessa matriz, geralmente algum algoritmo automatizado, o SIFT [18], por exemplo, é executado em busca dessas correspondências entre as duas diferentes imagens (algoritmos que realizam essa tarefa são chamados de *feature trackers*). Tal processo automatizado pode gerar irregularidades e incoerências nas correspondências entre pontos obtidos [17].

Devido a essas irregularidades, a obtenção da matriz fundamental a partir de oito correspondências quaisquer pode levar a uma matriz imprecisa, que não caracterize corretamente a cena. Para corrigir isto é necessário executar o *eight-point* em conjunção com algum algoritmo robusto para estimar uma matriz fundamental precisa.

Um dos métodos robustos mais utilizados é implementado através do algoritmo RANSAC (*RANdom SAmple Consensus*) [5]. O RANSAC é um algoritmo geral que gera iterativamente modelos a partir de um mínimo necessário de pontos selecionados aleatoriamente e mede a precisão do modelo para todos os outros pontos, classificando-os em *inliers* ou *outliers*. Assim, com várias iterações, o algoritmo é capaz de encontrar um modelo preciso para o problema em questão.

Outro fator de importância na busca da matriz fundamental é a possibilidade de obter uma estimação robusta dessa matriz em tempo real. A computação do RANSAC, como se verá, toma um grande tempo, por ser uma abordagem probabilística ao problema. Tal algoritmo pode ser facilmente paralelizado. Surge a idéia de utilizar as GPGPUs na implementação desses algoritmos, com o objetivo de diminuir o tempo de recuperação dessa matriz, otimizando o processo e possibilitando o uso do *pipeline* de reconstrução tridiomensional em sistemas que trabalhem em tempo real.

1.3 Objetivos

Este trabalho de graduação visa implementar o algoritmo de *eight-point*, utilizando-se do RANSAC para a obtenção de um resultado robusto. A implementação será realizada voltada para uma execução em GPU, utilizando-se CUDA, buscando por partes dos algoritmos que podem ser paralelizadas ao máximo. Utilizando-se do alto grau de eficiência das GPUs, obtém-se um maior desempenho com relação a suas implementações genéricas.

Para tal, um cuidadoso estudo sobre as etapas dos algoritmos que possam ser paralelizadas deve ser realizado. Geralmente, procura-se por etapas em que diversas computações são essencialmente independentes umas das outras, pois isso facilita a paralelização.

Como parte do estudo realizado, informações mais gerais sobre a recuperação da matriz fundamental, bem como de suas relações com outras etapas da reconstrução 3D e também das suas aplicações na área de visão computacional foram investigadas. Como subprodutos não menos importantes deste trabalho, serão criadas também em GPU algumas funções mais gerais importantes que são ferramentas fundamentais para os algoritmos citados acima. Tais funções têm grande importância não somente na área de visão computacional. Como exemplo, temos o SVD (*Singular Value Decomposition*) [10], necessário em algumas etapas

do *eight-point*, e que é utilizado em diversas aplicações na área de processamento de sinais e estatística.

Além do desenvolvimento do projeto em GPGPU, serão realizadas comparações com implementações genéricas já existentes, não apenas dos algoritmos em geral, mas também de suas sub-funções importantes citadas no parágrafo anterior. Entre as principais bibliotecas que já possuem os algoritmos, temos a VXL [14], a qual será fundamentalmente usada como a base para a comparação.

1.4 Estrutura do Trabalho

Este trabalho possui seis capítulos. O capítulo 2 fornece as ferramentas matemáticas básicas que serão utilizadas durante o desenvolvimento do conteúdo mais geral do texto, bem como fixa a notação que será utilizada.

O capítulo 3 discute detalhes dos algoritmos de *eight-point*, RANSAC e SVD. Estes algoritmos serão abordados do ponto de vista de uma implementação padrão, apenas tentando tornar claros quais os seus objetivos gerais e como eles os alcançam.

O capítulo 4 faz uma análise das implementações realizadas em GPU dos algoritmos descritos no capítulo 3. Inicialmente, o ambiente de programação CUDA é mais bem explicado. Então, será falado dos principais pontos de paralelização que foram explorados durante o desenvolvimento das soluções implementadas, bem como suas limitações.

No capítulo 5, os algoritmos implementados em GPGPU são comparados com as suas soluções genéricas implementadas em CPU.

Por fim, o capítulo 6 mostrará as principais conclusões obtidas deste trabalho de graduação, resumindo o que foi desenvolvido durante o mesmo.

2 Conceitos Preliminares

Este trabalho é fundamentado em idéias bastante consolidadas obtidas a partir de resultados matemáticos importantes da geometria projetiva. A importância da matemática para ele é, portanto, óbvia. Assim, este capítulo foi preparado com o intuito de introduzir as definições matemáticas básicas necessárias para um bom entendimento do texto, bem como convencionar a terminologia e notação que serão utilizadas posteriormente neste documento.

A maior parte dos conceitos explorados aqui requer um conhecimento prévio de álgebra linear, principalmente da idéia de vetores e como essas entidades matemáticas se relacionam, além de conhecimentos sobre transformações lineares e matrizes. Além disso, uma boa noção geométrica do que as entidades algébricas realmente querem transmitir é essencial. Convém ressaltar que a representação algébrica dos entes geométricos que é convencionada durante o desenvolvimento da geometria projetiva é um pouco distinta da apresentada num espaço vetorial como o $\mathbb{R}^n$, mas, pelo menos para o plano projetivo, será apresentado um modelo que facilita a sua interpretação.

O capítulo é dividido em três seções. A primeira traz os conceitos de geometria projetiva, grande parte deles utilizados logo em seguida na segunda seção. A segunda seção, que trata de geometria epipolar, é talvez a de maior importância para o trabalho, pois é exatamente onde este trabalho se encaixa, mostrando conceitos de grande importância, como o de retas epipolares e trazendo a idéia da matriz fundamental. Por último, temos uma curta seção que fala de uma decomposição de matrizes que facilita profundamente o desenvolvimento do algoritmo de *eight-point*, a decomposição em valores singulares (do inglês *Singular Value Decomposition*, SVD).

2.1 Geometria Projetiva

Aqui, será delineada grande parte das idéias que envolvem pontos, retas e planos no espaço projetivo. Inicialmente, falaremos em geral de um n-espaço projetivo e logo em seguida, daremos detalhes mais específicos para o plano projetivo e o 3-espaço projetivo. Durante o desenvolvimento desta seção foram utilizados como base principalmente [1], [7], [6].

2.1.1 n-Espaço Projetivo

Um ponto no n-espaço projetivo $\wp^n$ é representado por um vetor de $n+1$ coordenadas, não todas nulas. Assim, o ponto $\boldsymbol{x} = [x_1 \quad \cdots \quad x_{n+1}]^T$ é um ponto de $\wp^n$. As suas coordenadas são chamadas de coordenadas homogêneas. Em $\wp^n$, dizemos que os pontos $\boldsymbol{x}$ e $\boldsymbol{y}$ são iguais e escrevemos $\boldsymbol{x} \sim \boldsymbol{y}$ se, e somente se, existe um escalar λ diferente de zero, tal que $x_i = \lambda y_i$ para todo i entre 1 e $n+1$. Repare que dois vetores não necessitam necessariamente ter as mesmas coordenadas homogêneas para ser o mesmo ponto. Do ponto de vista de álgebra vetorial, $\boldsymbol{x} \sim \boldsymbol{y}$ equivale à igualdade $\boldsymbol{x} = \lambda \boldsymbol{y}$, para algum λ não nulo.

Uma projetividade ou colineação é uma transformação inversível h entre dois espaços projetivos que preserva a colinearidade. Isto significa que se h for uma transformação de $\wp^n$ em $\wp^m$, dois pontos que pertencem a uma mesma reta em $\wp^n$ serão mapeados em dois pontos de $\wp^m$ que também estão numa mesma reta. Pode ser provado que qualquer projetividade pode ser realizada através de uma multiplicação por uma matriz [1]. Assim, uma matriz $\boldsymbol{H}$de tamanho $(n+1) \times (m+1)$ representa uma projetividade $h: \wp^m \to \wp^n$, se tivermos $h(x) = \boldsymbol{H}\boldsymbol{x}$ para todo $\boldsymbol{x}$ do domínio. É importante ressaltar que as matrizes $\boldsymbol{H}$ e $\lambda \boldsymbol{H}$ representam a mesma homografia, pois, para todo $\boldsymbol{x}$, temos que $\lambda \boldsymbol{H}\boldsymbol{x} = \boldsymbol{H}(\lambda \boldsymbol{x}) \sim \boldsymbol{H}\boldsymbol{x}$.

A seguir, entraremos em detalhes mais específicos sobre os dois espaços projetivos de maior importância para este trabalho: o plano projetivo (2-espaço projetivo) e o 3-espaço projetivo.

2.1.2 Plano Projetivo

O plano projetivo pode ser definido simplesmente como o 2-espaço projetivo $\wp^2$. Assim, qualquer ponto em $\wp^2$ pode ser representado por um vetor de três coordenadas, com pelo menos uma delas diferente de zero. Por exemplo, o ponto $\boldsymbol{m}$ pode ser representado como $\boldsymbol{m} = [x \quad y \quad z]^T$ em coordenadas homogêneas.

Também iremos definir retas no plano projetivo. Uma reta, assim como um ponto, é representada por um vetor de três coordenadas, não todas iguais a zero. Temos então que a reta $l = [a \quad b \quad c]^T$. Da mesma maneira que ocorre com pontos, todos os múltiplos não nulos $\alpha l = [\alpha a \quad \alpha b \quad \alpha c]^T$ representam a mesma reta.

Dizemos que um ponto $\boldsymbol{m} = [x \quad y \quad z]^T$ está na reta $l = [a \quad b \quad c]^T$ se e somente se tivermos

$$l^T \boldsymbol{m} = 0.$$

Na equação acima, estamos realizando o produto escalar usual. Assim, um ponto está numa reta se e somente se o produto escalar entre seus vetores é nulo. Repare que se os pontos $\boldsymbol{x}$ e $\boldsymbol{y}$ são distintos, então teremos que a reta l que passa por eles é dada por $l = \boldsymbol{x} \times \boldsymbol{y}$, onde a operação $\times$ é o produto vetorial usual para vetores de três dimensões. Este resultado pode ser obtido trivialmente, notando-se que, $l^T \boldsymbol{x} = 0$ e $l^T \boldsymbol{y} = 0$.

Além da reta determinada por dois pontos, podemos falar do ponto determinado por duas retas. Se l e m são retas distintas de $\wp^2$, sendo $\boldsymbol{x}$ o ponto em que as duas passam ao mesmo tempo, temos que $l^T \boldsymbol{x} = 0$ e $m^T \boldsymbol{x} = 0$. Assim, analogamente ao resultado anterior, temos que o ponto x é dado por $\boldsymbol{x} = m \times l$.

A partir do resultado análogo obtido nos dois parágrafos anteriores, podemos enunciar um importante princípio para a geometria projetiva: o princípio da dualidade [6]. Sabemos que a operação de produto escalar entre dois vetores é uma operação comutativa. Assim, a definição de um ponto $\boldsymbol{m}$ estar em uma reta l equivale a de a reta l passar por um ponto $\boldsymbol{m}$. A simetria dada por essa definição implica que não há uma diferença formal entre pontos e retas.

A importância do princípio da dualidade é a que, para toda proposição no plano projetivo que envolva retas e pontos, existe uma proposição análoga a esta que envolve pontos e retas (isto é, se provarmos uma propriedade verdadeira, podemos trocar as expressões "ponto" e "reta" respectivamente por "reta" e "ponto", sem alterar a veracidade da proposição).

Uma determinada reta possui infinitos pontos. A afirmação dual a esta, seria que por um ponto passam infinitas retas. Na geometria projetiva, o conjunto das retas que passam por um determinado ponto é chamado de pencil de retas determinado por este ponto.

Para simplificar a notação, iremos representar produtos vetoriais através de multiplicação de matrizes utilizando o seguinte artifício: sendo $\boldsymbol{m} = [x \quad y \quad z]^T$ e $\boldsymbol{n}$, pontos quaisquer de $\wp^2$, temos que $\boldsymbol{m} \times \boldsymbol{n} = [\boldsymbol{m}]_\times \boldsymbol{n}$, sendo $[\boldsymbol{m}]_\times = \begin{bmatrix} 0 & z & -y \\ -z & 0 & x \\ y & x & 0 \end{bmatrix}$.

2.1.2.1 Representação do plano projetivo

As idéias vistas do ponto de vista algébrico na seção anterior podem ser melhor absorvidas aqui, onde será feito um tratamento do modelo de plano afim estendido para a representação geométrica do plano projetivo.

Vamos supor que temos um observador centrado na origem do $\mathbb{R}^3$, olhando na direção do eixo z, sentido positivo. Aqui, consideraremos o plano afim $z = 1$ do espaço tridimensional, como sendo o plano no qual as imagens do $\mathbb{R}^3$ são projetadas. Assim, todas as retas que passam pela origem, representam alguma imagem projetada.

Repare que um ponto qualquer $[x \quad y]^T$ do plano afim $(z = 1)$ pode ser representado como o ponto $[x \quad y \quad 1]^T$ do $\mathbb{R}^3$. Na visão do plano projetivo, tal ponto pode ser identificado com as coordenadas homogêneas $[\alpha x \quad \alpha y \quad \alpha]^T$para qualquer α não nulo.

Sabemos que no espaço tridimensional, $\alpha[x \quad y \quad z]^T$ para valores de α variando de $-\infty$ a ∞ representa uma reta que passa pela origem e que possui a direção do vetor $[x \quad y \quad z]^T$. Logo, todo ponto projetivo de $\wp^2$ equivale a uma reta em $\mathbb{R}^3$. A figura Figura 2.1 ilustra essa situação. **Podemos então concluir que o ponto projetivo** $[x \quad y \quad z]^T$ corresponde ao ponto afim $[x/z \quad y/z]^T$ (para z não nulo).

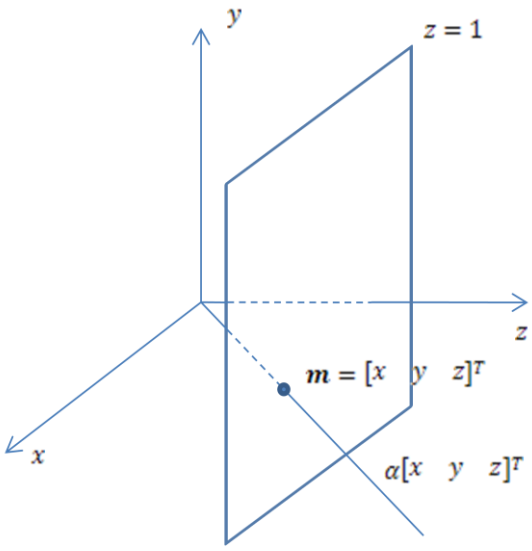

Figura 2.1. Reta $\alpha[x \quad y \quad z]^T$ em $\mathbb{R}^3$ e a sua representação m no plano afim $z = 1$.

Quanto à representação de retas projetivas no plano afim, temos que, no $\mathbb{R}^3$, um plano que passa pela origem e que possui como vetor normal $l = [a \quad b \quad c]^T$, tem equação cartesiana $ax + by + cz = 0$. Podemos ver a representação da reta projetiva que possui tais

coordenadas homogêneas no plano afim $z = 1$, como sendo a reta determinada pela sua interseção com esse plano. A figura Figura 2.2 ilustra esse fato.

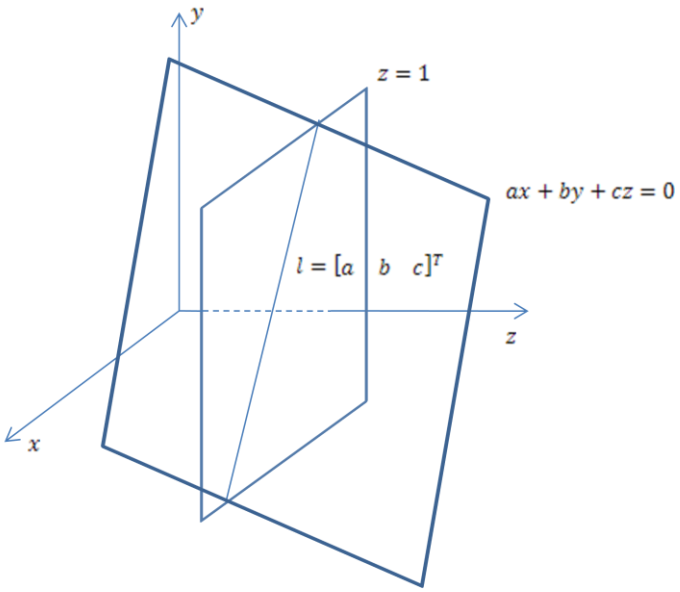

Figura 2.2. Representação no plano afim da reta projetiva *l*, como a interseção entre o plano $z = 1$ e o plano $ax + by + cz = 0$.

É interessante notar que, um ponto projetivo $\boldsymbol{m} = [x \quad y \quad z]^T$ está na reta projetiva $l = [a \quad b \quad c]^T$ se e somente se o plano $ax + by + cz = 0$ contém a reta paramétrica $\alpha[x \quad y \quad z]^T$, como pode ser visto na Figura 2.3.

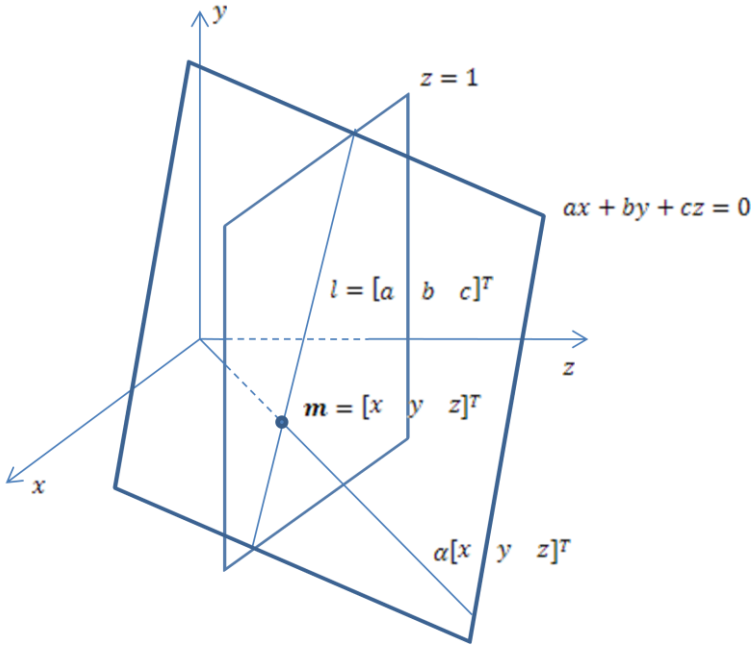

Figura 2.3. O ponto projetivo *m* está na reta projetiva *l*.

Um último tópico a ser tocado nesta seção é a questão da reta no infinito. Sejam as retas projetivas $l = [a \quad b \quad c]^T$ e $m = [a \quad b \quad d]^T$. Repare que o ponto determinado por essas duas retas tem sua terceira componente nula. Nesse caso, não podemos representá-lo no plano afim (pois não podemos dividir por zero). Assim, dizemos que essas retas se encontram num ponto no infinito. Repare que todo ponto no infinito está na reta $l_\infty = [0 \quad 0 \quad 1]^T$, pois, qualquer que seja o ponto $\boldsymbol{x} = [p \quad q \quad 0]$, tem-se que ${l_\infty}^T x = 0$. Tal reta é chamada de reta no infinito.

2.1.3 3-Espaço Projetivo

Um ponto no 3-espaço projetivo $\wp^3$ pode ser visto como um vetor de quatro coordenadas homogêneas. Assim, $\boldsymbol{X} = [X \quad Y \quad Z \quad W]$ é um ponto projetivo em $\wp^3$. A dualidade no 3-espaço projetivo ocorre entre pontos e planos, os quais são também representados por vetores de quatro coordenadas [6]. Analogamente ao $\wp^2$, dizemos que um ponto $\boldsymbol{X}$ pertence ao plano Π, se e somente se tivermos

$$\Pi^{\mathrm{T}}\mathbf{X} = 0.$$

Também existem modelos para representar o 3-espaço projetivo. Analogamente ao modelo afim estendido em $\wp^2$, o principal modelo que o representa leva em conta a projeção de hiperplanos, planos e retas pela origem do $\mathbb{R}^4$ no hiperplano afim $W = 1$.

2.2 Modelo de Câmera

Antes de começar a falar sobre geometria epipolar, esta seção introduz um modelo de câmera rudimentar, para ter-se uma idéia de como imagens digitais podem ser obtidas. Esta seção não visa tratar um modelo completo para uma câmera e sim apenas introduzir a idéia de projeção de coordenadas de mundo para coordenadas de câmera. Isto será assim, pois para a continuidade do texto não são necessários conhecimentos maiores sobre este assunto. Para uma idéia menos introdutória a modelos de câmera, veja [7]. Para um tratamento mais profundo, recomendo [1].

Nesse trabalho, será apresentada uma introdução ao modelo de câmera perspectivo que é utilizado em [7]. Nesse modelo, o processo de formação de uma imagem é completamente determinado quando se conhece o centro de projeção $\mathbf{C}$ da câmera e o seu plano de projeção

$\mathcal{R}$. O processo de projeção é encarado como uma transformação que leva os pontos de coordenadas de mundo para coordenadas de imagem.

Quando **C** está localizado na origem e seu plano de projeção é $Z = 1$ (distância focal unitária), sendo (x, y) as coordenadas de imagem de um ponto que em coordenadas de mundo é dado por (X, Y, Z) são dadas por $x = \frac{X}{Z}$ e $y = \frac{Y}{Z}$. Tal transformação pode ser representada pela seguinte multiplicação matricial:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}.$$

A Figura 2.4 ilustra a idéia de projeção em câmera.

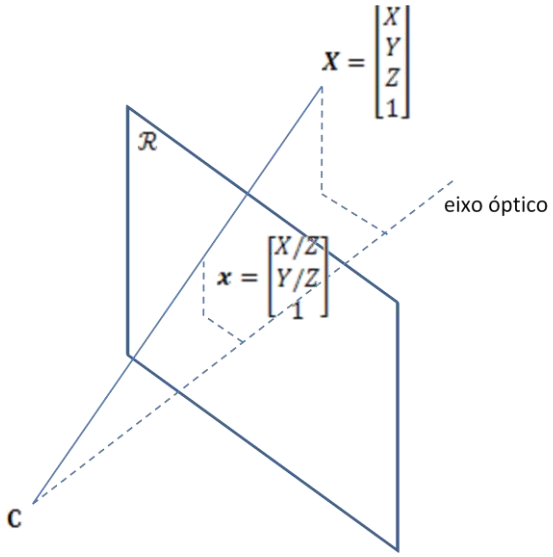

Figura 2.4 - Projeção em câmera.

Para desenvolver um modelo de câmera que leve em conta uma localização distinta da origem e movimentações, bem como sua calibração intrínseca (distância focal não unitária, informações sobre os *pixels*, etc.), precisaríamos desenvolver maiores detalhes num estudo sobre modelos de câmera. Deixo claro, que os pontos citados acima que aqui não estão sendo levados em conta são modelados também através de transformações que podem ser realizadas através de simples multiplicações matriciais. Porém, isto não será necessário, já que o algoritmo de *eight-point* em si não precisa dos parâmetros de calibração da câmera para a obtenção da matriz fundamental (isto é, o algoritmo pode ser utilizado mesmo em câmeras descalibradas).

2.3 Geometria Epipolar

Nesta seção, será introduzida a geometria projetiva de duas visualizações. A ela, damos o nome de geometria epipolar. É natural que existam relações entre duas imagens de uma mesma cena que foram obtidas a partir de diferentes câmeras. Por esse trabalho lidar com a recuperação da matriz fundamental, essa seção é de extrema importância e explica qual a relação entre o ponto de uma imagem e seu correspondente em outra imagem da mesma cena.

2.3.1 Definições básicas

Vamos iniciar nosso tratamento de geometria epipolar definindo alguma terminologia básica. Seja $\boldsymbol{X}$ um ponto no 3-espaço do qual foram obtidas duas imagens a partir de diferentes visualizações. Seja $\mathbf{C}$ o centro de projeção da primeira câmera e $\mathbf{C}'$ o da segunda. Seja $\boldsymbol{x}$ a projeção de $\boldsymbol{X}$ na primeira dessas imagens e $\boldsymbol{x}'$ sua projeção na segunda imagem. Chamamos de *baseline* a reta que passa pelos centros de projeções das duas câmeras.

A *baseline* determina nos dois planos de projeção (imagem) de cada câmera os pontos $\boldsymbol{e}$ e $\boldsymbol{e}'$, chamados de epipolos. Qualquer plano que passa pela *baseline* é denominado plano epipolar. Um ponto $\boldsymbol{X}$ e os centros de projeção $\boldsymbol{C}$ e $\boldsymbol{C}'$ determinam um plano epipolar Π. As projeções $\boldsymbol{x}$ e $\boldsymbol{x}'$ de $\boldsymbol{X}$ estão em Π. A interseção de um plano epipolar com o plano de alguma das imagens determina a chamada reta epipolar (l para a primeira imagem e l′ para a segunda). A Figura 2.5 - Elementos da Geometria Epipolar. ilustra todos os conceitos vistos até aqui.

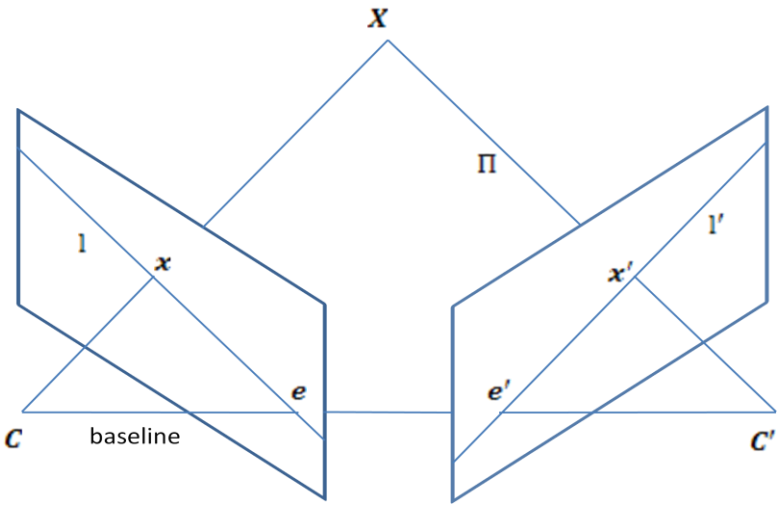

Figura 2.5 - Elementos da Geometria Epipolar.

Repare que, para cada ponto $\boldsymbol{x}$ da primeira imagem, existe uma reta epipolar l' na segunda imagem. Equivalentemente, no sentido inverso, para cada ponto $\boldsymbol{x}'$, existe uma reta epipolar l na primeira imagem. Na verdade, tal correspondência é uma projetividade que leva pontos a retas. A matriz fundamental é a matriz que caracteriza essa projetividade.

Agora que os conceitos básicos foram ilustrados, falaremos da matriz fundamental.

2.3.2 A matriz fundamental

A matriz fundamental é uma matriz 3×3 de posto 2 que incorpora as informações sobre a orientação relativa entre as câmeras e suas respectivas geometrias internas [1], [4]. Como definido na seção anterior, se $\boldsymbol{x}$ for o vetor de coordenadas homogêneas de um ponto numa imagem e $\boldsymbol{x}'$ o vetor de coordenadas do mesmo ponto numa segunda imagem, a matriz fundamental F satisfaz a restrição epipolar:

$$\boldsymbol{x}'^{T} F \boldsymbol{x} = 0.$$

Vê-se então a importância da matriz fundamental na caracterização da geometria de duas visões: podemos computá-la através de correspondências $\boldsymbol{x_i}$ e $\boldsymbol{x_i'}$, sem a necessidade de obter os parâmetros das câmeras.

Aqui será mostrada a derivação geométrica da matriz fundamental presente em [1]. Essa derivação é baseada em dois passos: inicialmente, mapeamos o ponto $\boldsymbol{x}$ de uma imagem num ponto $\boldsymbol{x}'$ da segunda através da re-projeção por algum plano. Então, a reta determinada por esse ponto e o epipolo da segunda imagem é então a reta epipolar relativa a $\boldsymbol{x}$. A Figura 2.6 ilustra essa transferência por plano.

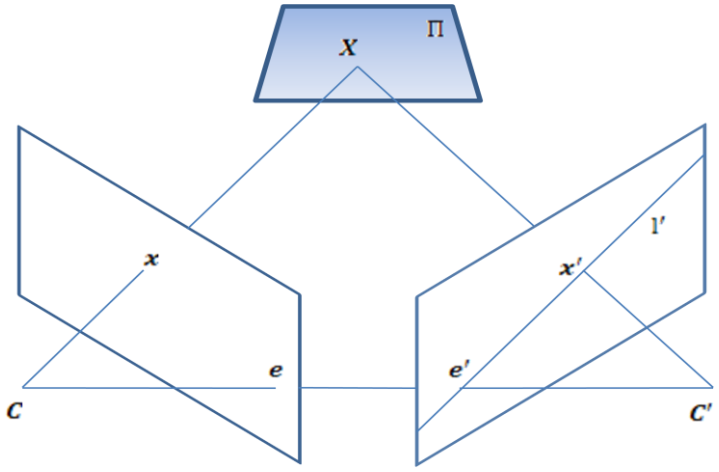

Figura 2.6 - Transferência por plano.

Em mais detalhes, consideremos um plano Π que não contenha os centros de projeção das câmeras. A partir da reta determinada por $\boldsymbol{C}$ e $\boldsymbol{x}$ encontramos o ponto $\boldsymbol{X}$, a interseção do plano Π com essa reta. Projetamos então este ponto $\boldsymbol{X}$ na segunda imagem, obtendo o ponto $\mathbf{x}'$. Pode-se mostrar que existe uma projetividade que mapeia $\boldsymbol{x}$ em $\boldsymbol{x}'$ [1]. Chamaremos essa homografia de H_π. Assim, $\boldsymbol{x}' = H_\pi \boldsymbol{x}$.

Após a re-projeção do ponto $\boldsymbol{X}$, a reta epipolar l' pode ser obtida usando-se a definição de reta que passa por ponto: $l' = \boldsymbol{e}' \times \boldsymbol{x}' = [\boldsymbol{e}']_\times \boldsymbol{x}'$. Usando o resultado do parágrafo anterior, obtemos que $l' = [\boldsymbol{e}']_\times \boldsymbol{x}' = [\boldsymbol{e}']_\times H_\pi \boldsymbol{x}$. Dizemos então que $F = [\boldsymbol{e}']_\times H_\pi$ é a matriz fundamental dessa geometria. Note que como $[\boldsymbol{e}']_\times$ é uma matriz de posto 2, F tem posto 2.

Embora uma derivação algébrica da matriz fundamental possa ser obtida a partir dos parâmetros de câmera, a abordagem desse trabalho não necessita de uma derivação como essa. Isto é justificado pelo próprio modo como o algoritmo de *eight-point* ataca o problema, isto é, utilizando apenas correspondências entre imagens, sem a necessidade dos parâmetros de câmera.

A principal propriedade da matriz fundamental, antecipada no primeiro parágrafo, pode ser facilmente demonstrada agora. Sabemos que $l' = F\boldsymbol{x}$. Como $\boldsymbol{x}'$ deve pertencer à reta l', temos trivialmente que $\boldsymbol{x}'^T l' = 0$ e obtemos $\boldsymbol{x}'^T F \boldsymbol{x} = 0$. Outro fato que merece ser atentado é que a matriz fundamental que leva os pontos $\boldsymbol{x}'$ da segunda imagem às retas l da primeira é exatamente a transposta da matriz fundamental que leva $\boldsymbol{x}$ a l', isto é, se $l' = F\boldsymbol{x}$, então $l = F^T \boldsymbol{x}'$.

Mais será dito a frente com relação à matriz fundamental, principalmente no que condiz ao algoritmo de *eight-point*.

2.4 Decomposição em Valores Singulares (SVD)

A decomposição em valores singulares é uma das mais robustas e importantes formas de fatoração de uma matriz, que revela muito de sua estrutura e que possui diversas aplicações em diversas áreas, como processamento de sinais e estatística [8]. É interessante deixar claro que a definição de SVD não requer que M seja uma matriz real, mas para os propósitos desse trabalho, isto é suficiente.

Seja M uma matriz real $m \times n$ qualquer. Então, podemos escrever M como:

$$M = U\Sigma V^T.$$

Sendo U uma matriz $m \times m$ ortonormal, Σ uma matriz diagonal $m \times n$ com números não negativos em sua diagonal chamados de valores singulares e V uma matriz $n \times n$ ortonormal. Essa fatoração da matriz M é a decomposição em valores singulares. Os valores singulares na matriz Σ são geralmente colocados em ordem decrescente da primeira linha para a última.

Vale citar que o importante problema de se minimizar a norma $\|M\boldsymbol{x}\|$ com relação ao vetor $\boldsymbol{x}$, sob a restrição de que $\|\boldsymbol{x}\| = 1$ é de fácil resolução utilizando-se a decomposição SVD: sua solução é dada pelo vetor coluna de V relativo ao menor valor singular, que, no caso dos valores singulares estarem colocados em ordem decrescente é o último vetor coluna de V.

A importância dessa decomposição para este trabalho é a de que ela é utilizada no algoritmo de *eight-point*. Quando delinearmos este algoritmo, serão apontados dois pontos do algoritmo que são facilitados pelo SVD: encontrar uma solução de mínimos quadrados e restringir o posto da matriz fundamental encontrada.

A obtenção do SVD do ponto de vista algorítmico será mais tarde elucidada no capítulo relativo aos algoritmos implementados, já que uma versão do algoritmo em [9] implementada em CUDA será apresentada. Mais detalhes sobre essa decomposição podem ser encontrados em [10].

3 Algoritmos

Neste capítulo, serão apresentados e discutidos os três principais algoritmos observados neste trabalho: *eight-point*, RANSAC e SVD. Serão vistos os seus objetivos gerais, e como eles são estruturados para alcançar este objetivo. Além disto, detalhes técnicos de suas implementações padrão serão ressaltados.

Teremos três seções, uma relativa a cada um dos algoritmos. Iniciamos falando do *eight-point* na primeira seção. Veremos como o algoritmo consegue gerar, a partir de no mínimo oito correspondências entre duas imagens de uma cena, a matriz fundamental. A segunda seção é relativa ao algoritmo RANSAC e busca explicar como ele estima com robustez modelos precisos. Na última seção, falarei de como pode ser realizada uma implementação em paralelo do SVD, que será de grande importância para o algoritmo de *eight-point*.

3.1 *Eight-point*

A matriz fundamental de uma cena é uma necessidade que deve ser atendida para realizar a reconstrução 3D desta cena a partir de duas imagens obtidas de diferentes câmeras. Como geralmente os parâmetros das câmeras que obtiveram as imagens são desconhecidos (isto é, as câmeras são descalibradas), a matriz fundamental deve ser computada a partir de restrições obtidas a partir das imagens. O *eight-point* é um algoritmo rápido e de fácil implementação que serve a esse propósito.

A idéia principal do algoritmo de *eight-point* foi introduzida em 1981 num artigo que hoje é considerado um clássico [2]. Esta solução inicial, porém, é extremamente suscetível a ruído e incoerência nos dados, sendo vista como inferior a outros algoritmos que computam a matriz fundamental [3]. [3] mostra uma variante no algoritmo de *eight-point*, conhecida como *eight-point* normalizado, em que as coordenadas dos pontos das imagens são transladadas e escaladas. Essa variante é a que será mostrada aqui neste texto, devido a sua maior robustez.

3.1.1 Estimando a matriz fundamental

Um mesmo ponto 3D de uma cena projetado no plano de projeção de duas câmeras distintas dá origem a um par de pontos correspondentes entre as duas imagens geradas. Estes pontos têm as coordenadas homogêneas $\boldsymbol{x} = [x_1 \quad x_2 \quad 1]^T$ em um primeiro plano de projeção

e $\boldsymbol{x}' = [x_1' \quad x_2' \quad 1]^T$ no outro. Sendo F a matriz fundamental que caracteriza a geometria epipolar da cena, sabemos que esses pontos obedecem à restrição epipolar

$$\boldsymbol{x}'^T F \boldsymbol{x} = 0.$$

O objetivo do algoritmo de *eight-point* é então estimar a matriz F. Dado um conjunto de entrada composto por pontos de correspondências entre as duas imagens, vamos inicialmente estimar uma matriz que seja compatível com um determinado número n de correspondências, então vamos tornar a matriz fundamental estimada uma matriz de posto 2, como exigido pela sua definição.

Como temos n correspondências entre as imagens, que são dadas pelos pares $(\boldsymbol{x}_i, \boldsymbol{x}_i') = ([x_{i1} \quad x_{i2} \quad 1]^{\mathrm{T}}, [x_{i1}' \quad x_{i2}' \quad 1]^T)$, com $i \in \{1, \dots, n\}$, e cada um desses pares dá origem a uma equação linear nos termos da matriz F, podemos escrever o seguinte sistema de n equações, cuja matriz fundamental a ser estimada deve obedecer:

$$\begin{cases} \boldsymbol{x}_1'^T F \boldsymbol{x}_1 = 0 \\ \boldsymbol{x}_2'^T F \boldsymbol{x}_2 = 0 \\ \vdots \\ \boldsymbol{x}_n'^T F \boldsymbol{x}_n = 0 \end{cases}$$

Escrevendo a matriz F de modo a deixar explícitos seus termos:

$$F = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}.$$

Podemos reescrever o sistema de equações acima na forma matricial:

$$\begin{bmatrix} x_{11}x_{11}' & x_{12}x_{11}' & x_{11}' & x_{11}x_{12}' & x_{12}x_{12}' & x_{12}' & x_{11} & x_{12} & 1 \\ x_{21}x_{21}' & x_{22}x_{21}' & x_{21}' & x_{21}x_{22}' & x_{22}x_{22}' & x_{22}' & x_{21} & x_{22} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{n1}x_{n1}' & x_{n2}x_{n1}' & x_{n1}' & x_{n1}x_{n2}' & x_{n2}x_{n2}' & x_{n2}' & x_{n1} & x_{n2} & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = \mathbf{0}.$$

O sistema acima, representado de agora em diante simplesmente por $A\boldsymbol{f} = \mathbf{0}$, possui n equações nas nove incógnitas que são os termos da matriz fundamental. Por a matriz fundamental ser equivalente a qualquer múltiplo dela mesma, o grau de liberdade na verdade

é de oito parâmetros [1]. Assim, para resolver este sistema, precisamos de no mínimo oito equações (isto é, oito correspondências). O *eight-point* trabalha resolvendo este sistema baseado nessa premissa, daí, seu nome (na verdade, o número mínimo de equações é sete, por causa da restrição de posto 2. O *eight-point* lida com isso mudando o posto da matriz final).

Geralmente, podemos obter uma solução de mínimos quadrados para o sistema linear obtido. Tal solução pode ser obtida minimizando $\|A\boldsymbol{f}\|$, sujeito a condição de que $\|\boldsymbol{f}\| = 1$ (condição que garante que $\boldsymbol{f}$ não seja o vetor nulo). Como visto na seção 2.4, tal problema pode ser solucionado realizando-se a computação do SVD da matriz A. A solução é um vetor $\boldsymbol{f}$, que após ser reorganizado, em forma de matriz, dá origem a matriz F.

A matriz F obtida dessa maneira possui geralmente posto 3. Devemos impor então a restrição de posto 2. Podemos utilizar novamente a decomposição SVD com tal objetivo [1]. Para isso, obtemos o SVD da matriz F e fazemos a imposição de que seu menor valor singular é nulo, zerando-o. A matriz F é então reconstruída através do produto das matrizes de sua decomposição. Ela possuirá posto 2 e será a matriz fundamental estimada pelo *eight-point*.

3.1.2 Normalização

A normalização das coordenadas dos pontos de imagem é executada previamente à estimação da matriz F, aumentando a precisão do resultado final, como previsto em [4]. Ao final da estimação da matriz, obtemos uma matriz fundamental para as coordenadas normalizadas. Realizamos então a denormalização dela, obtendo a matriz fundamental F nas coordenadas originais. Mais detalhes sobre essa normalização podem ser encontrada no artigo citado.

No quadro abaixo, segue o algoritmo de *eight-point* sintetizado:

Eight-point

- Selecionar 8 (ou mais) pontos e os normalizar.
- Encontrar a solução de mínimos quadrados que obedece a restrição epipolar ($x'Fx = 0$), usando-se SVD.
- Forçar o posto da matriz a ser 2, zerando o menor valor singular, usando-se a SVD.
- Denormalizar a matriz fundamental.

3.2 RANSAC

O RANSAC (*RANdom SAmple Consensus*) é um estimador robusto de parâmetros a partir de dados para modelos matemáticos. Ele foi inicialmente proposto em um artigo em 1981 [5].

A principal vantagem no uso do RANSAC nesse trabalho é que ele possui a propriedade de ser um estimador robusto. Estimadores de parâmetros clássicos, como mínimos quadrados, se baseiam em todos os dados que tomam como entrada. Desse modo, se grande parte dos dados vier de uma fonte em que haja erros inerentes, ou ruidosos, (isto é, algum tipo de *outlier*) o modelo final estimado será prejudicado. O RANSAC, porém, foi desenvolvido com o intuito de evitar que *outlier*s atrapalhem a computação do modelo final.

Repare que nessa discussão, a palavra ponto não necessariamente se refere à primitiva geométrica de localização. Ela pode se referir a um determinado dado de um conjunto de entrada. O uso da palavra ficará claro do contexto.

3.2.1 O algoritmo

No RANSAC, inicialmente amostras mínimas de dados são selecionadas aleatoriamente. Uma amostra mínima depende do modelo a ser gerado. Geralmente, é o número mínimo n de dados que será utilizado para inicializar um modelo. Por exemplo, no caso de o modelo a ser estimado ser uma reta, uma amostra mínima consiste de apenas dois elementos.

A partir de uma amostra mínima, um modelo é então instanciado. O restante dos dados será utilizado apenas para validar este modelo. Para tal, é definido um valor de limiar t, que é a distância máxima do modelo que um dado pode estar para ainda ser considerado um *inlier*. O conjunto de *inliers* associado a um modelo é o conjunto de consenso (*consensus set*) do modelo.

O tamanho do conjunto de consenso é o que determina o quão bom um modelo é. Geralmente, um parâmetro T do algoritmo define o número necessário que deve ser alcançado para um modelo ser considerado interessante. Quando algum dos modelos gerados iterativamente durante a execução do RANSAC obtiver mais de T *inliers*, o algoritmo finaliza, gerando novamente um modelo utilizando todos os T *inliers* obtidos, que será o modelo final encontrado pelo algoritmo (refinamento do modelo). Obviamente, é necessário

definir um número máximo N de iterações que o algoritmo irá realizar, para evitar um número infinito de iterações.

Se um determinado modelo não alcançar T *inliers*, o algoritmo gera um novo modelo a partir de uma nova seleção de uma amostra mínima dos dados de entrada. O algoritmo prossegue até conseguir obter um modelo interessante, ou alcançar o número máximo de iterações. No caso de a execução do algoritmo finalizar por alcançar o número máximo de iterações, pode-se retornar o modelo com mais *inliers* (na verdade, é retornado um novo modelo baseado em todos os *inliers*, um refinamento do obtido), ou retornar que o algoritmo falhou.

3.2.2 Parâmetros

Os três principais parâmetros que podem ser alterados de modo a determinar um bom resultado para o RANSAC são t, T e N. Uma breve discussão sobre eles é dada aqui. Mais detalhes podem ser encontrados em [5], [1].

O parâmetro t é o limiar de distância que uma determinada entrada pode estar do modelo para ser considerado um *inlier*. Geralmente este limiar depende da aplicação que está utilizando o modelo e dos propósitos a serem alcançados, portanto, não há uma solução muito geral para se encontrar o seu valor. Um estudo sobre a distribuição de probabilidade dos dados pode ser realizado para determinar um bom valor para t. Na prática, esse valor é determinado empiricamente [1].

Quanto ao valor de N, o número máximo de iterações que devem ser realizadas, um simples cálculo baseado na teoria das probabilidades pode ser realizado para determinar um valor justificável para ele. Queremos que, com probabilidade p, pelo menos uma das amostras mínimas de n pontos não possua *outliers*. Assim, p de certa forma é a probabilidade de que o algoritmo produza um resultado bom. Seja q a probabilidade de um dos dados ser um *inlier* do modelo. Repare que $(1 - q^n)$ indica a probabilidade de que ao menos um destes pontos seja um *outlier*. Como N é o número de vezes que o algoritmo vai iterar, isso significa que $(1 - q^n)^N = 1 - p$. Daí, obtemos que o número de iterações necessárias é dado por

$$N = \frac{\log(1-p)}{\log(1-q^n)}.$$

Durante a execução do algoritmo, podemos utilizar a porcentagem de *inliers* do melhor modelo (o que possuir mais *inliers*) obtido até agora como a probabilidade q para atualizar o valor de N adaptativamente. O valor de p geralmente é escolhido como 99%.

O parâmetro T também tem grande importância na determinação de um bom modelo. É natural que um modelo não precisa possuir 99% de *inliers* para ser considerado um modelo interessante. Um valor bom, porém, depende de cada caso e é geralmente determinado empiricamente.

3.2.3 RANSAC com *eight-point*

O modelo que pretendemos obter com o algoritmo de *eight-point* é a matriz fundamental. O número de correspondências mínimas necessárias para gerar um modelo utilizando o *eight-point* é de oito correspondências. Sendo assim, a amostra mínima geralmente possui oito elementos.

Quando se usa o RANSAC, a distância ao modelo deve ser definida para podermos decidir se um determinado ponto é *inlier* ou não do modelo. Existem algumas maneiras distintas de se definir a distância de uma correspondência de pontos entre duas imagens. A maneira que aqui será utilizada é chamada de distância epipolar simétrica [1]. No caso da matriz fundamental, podemos definir tal distância como a soma das distâncias dos pontos às retas epipolares às quais eles deveriam pertencer (no caso ideal, essa distância deve ser nula). Assim, se os pontos $\boldsymbol{x}$ e $\boldsymbol{x}'$ se correspondem em duas imagens, e a matriz F foi estimada como a matriz fundamental dessa geometria, temos que, $F\boldsymbol{x} = l'$ e $F^T\boldsymbol{x}' = l$ são as retas epipolares correspondentes. A medida de distância será dada por $d = d(\boldsymbol{x}, l) + d(\boldsymbol{x}', l')$, onde $d(\boldsymbol{x}, l)$ é a distância entre o ponto $\boldsymbol{x}$ e a reta l.

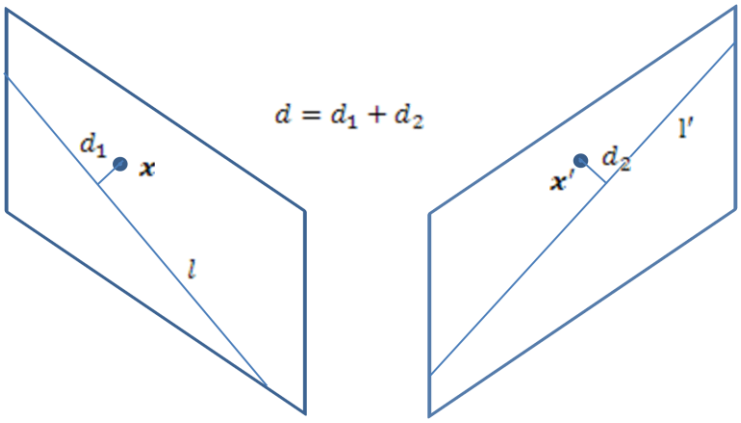

Figura 3.1 - Distância epipolar simétrica.

De modo a tornar claro o que foi apresentado, podemos ter uma idéia melhor sobre o significado dessa distância observando a Figura 3.1 - Distância epipolar simétrica..

Por fim, apresento o quadro abaixo, o qual resume o algoritmo de RANSAC, focando no caso em que o modelo a ser estimado é a matriz fundamental.

RANSAC

- Selecionar, de *N* pontos, *k* aleatoriamente.
- Gerar um modelo (matriz fundamental) que passe por esses *k* pontos.
- Avaliar, baseado num limiar *t* e numa distância definida, quantos dos *N* pontos estão no modelo(*inliers*).
- Repetir os passos anteriores até achar um modelo bom o bastante ou até um limite.

3.3 SVD

Existem diversos algoritmos que se propõem a computar a decomposição SVD de uma matriz. A implementação padrão encontrada em diversas bibliotecas de álgebra linear (entre elas a LINPAC [11]) é o algoritmo de SVD de Golub- Reinsch [12].

Por estar desenvolvendo um algoritmo que será utilizado num ambiente de computação extremamente paralelizado, o SVD aqui utilizado será baseado numa técnica exposta no artigo [9]. Nela, a computação do SVD é realizada numa estrutura de *systolic array*, usando o método Kogbetliantz para rotação bilateral, os quais são facilmente paralelizáveis [9]. Além de facilmente paralelizável, esse método é muito usado por ser numericamente estável [8].

3.3.1 O método de Kogbetliantz

O método consiste em multiplicar a matriz de entrada à esquerda e à direita de modo a zerar os elementos fora da diagonal principal. Foi demonstrado em [8] que o problema do SVD pode ser resolvido resolvendo-se uma seqüência de SVDs de matrizes 2×2. Mais detalhes sobre o método podem ser encontrados em [9].

As matrizes que multiplicam a esquerda e a direita são referidas pela notação $J^L(p,q,\theta)$ e $J^R(p,q,\phi)$, respectivamente. Os valores inteiros p e q indicam qual par de colunas está sendo diagonalizado no momento. Essas matrizes são construídas de modo a afetar somente essas duas colunas por vez, como veremos. É isso que permite a computação em paralelo.

Sendo A a matriz de entrada, com uma escolha apropriada das matrizes a serem multiplicadas à esquerda e à direita podemos realizar o seguinte conjunto de operações de modo à diagonalizar A:

$$A_0 = A$$
$$A_{k+1} = J_k^L(p,q,\theta)^T \, A_k \, J_k^R(p,q,\phi)$$

A matriz A_{k+1} irá convergir para a matriz de valores singulares Σ do SVD. O produto das matrizes multiplicadas à esquerda pode ser utilizado para obter a matriz V do SVD. O mesmo acontece com a matriz U, em relação às matrizes multiplicadas à direita. Assim, obtemos a decomposição SVD da matriz de entrada:

$$\Sigma = A_{k+1}$$
$$\mathrm{V} = \prod_{\mathrm{i}} J_i^L$$
$$\mathrm{U} = \prod_{\mathrm{i}} J_i^R$$

As matrizes de rotação $J(p,q,\phi)$, são matrizes identidades, exceto pelos elementos $J_{pp} = \cos\phi$, $J_{pq} = \sin\phi$, $J_{qp} = -\sin\phi$ e $J_{qq} = \cos\phi$. Tais elementos são os responsáveis pela ortogonalização das colunas p e q:

$$J(p,q,\phi) = \begin{bmatrix} 1 & & & & & & 0 \\ & \ddots & & \vdots & & \iddots & \\ & & \cos\phi & 0 & \sin\phi & & \\ & \cdots & 0 & 1 & 0 & \cdots & \\ & & -\sin\phi & 0 & \cos\phi & & \\ & \iddots & & \vdots & & \ddots & \\ 0 & & & & & & 1 \end{bmatrix}.$$

Podemos calcular separadamente o valor do ângulo θ e ϕ baseados no problema simples para matrizes 2×2. Para tal, devemos encontrar ângulos θ e ϕ de modo que

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}^T \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}.$$

Dada a matriz $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, encontrar os ângulos θ e ϕ não é difícil. Embora métodos que melhoram a precisão do algoritmo sejam preferíveis, podemos simplesmente resolver o sistema de equações

$$\begin{cases} \phi + \theta = \tan^{-1}\frac{c+b}{d-a} \\ \phi - \theta = \tan^{-1}\frac{c+b}{d-a}. \end{cases}$$

Então, para construir as matrizes $J^L(p,q,\theta)$ e $J^R(p,q,\phi)$, podemos computar os valores de θ e ϕ a partir dos elementos A_{pp}, A_{pq}, A_{qp} e A_{qq}, como se fizessem parte da matriz 2×2 que queremos diagonalizar.

A cada iteração, um par de colunas (p,q) é ortogonalizado. Para paralelizar ao máximo as operações, podemos definir uma ordenação em que os pares ordenados (p,q) produzirão as matrizes de rotação, sem as operações de colunas e linhas distintas interferirem na de outras colunas e linhas. Tal ordenação é sugerida em [9].

3.3.2 Implementação em *Systolic Array*

Um *systolic array* é uma estrutura composta de unidades de processamento interconectadas. [9] faz uso da estrutura de um *systolic array* para produzir um algoritmo em paralelo baseado no método descrito na seção anterior.

A estrutura utilizada no artigo visa encontrar o SVD de uma matriz $n \times n$ utizando uma estrutura de *array* com $\frac{n}{2} \times \frac{n}{2}$ células de processamento. Na prática, SVD de matrizes $m \times n$ também podem ser computados com o método, apenas completando a matriz de forma a ela virar uma matriz quadrada, com elementos iguais a um na diagonal principal e zero nos demais.

Cada uma das células de processamento do *array* será responsável por tratar submatrizes 2×2 da matriz de entrada. Essas células são distinguidas entre células da diagonal, que são responsáveis por elementos pelas submatrizes que estão na diagonal principal da matriz; e células fora da diagonal, que são responsáveis pelo restante das submatrizes.

As células da diagonal calculam os valores dos ângulos de rotação que diagonalizam a sua submatriz. Elas são então responsáveis por repassar os valores do ângulo de rotação para as outras células da mesma coluna e mesma linha, as quais realizam as operações de rotação utilizando os ângulos recebidos.

Quando todas as células realizam a operação de rotação, os elementos das matrizes são trocados, simulando uma troca de colunas da matriz de entrada, de modo ao cálculo dos ângulos de rotação serem computados utilizando outras colunas a serem ortogonalizadas como explicado no último parágrafo da subseção anterior.

Ao fim de uma varredura destas, o procedimento é realizado novamente, até obter-se uma matriz diagonal, com a precisão desejada. As matrizes de rotação podem ser guardadas durante o procedimento e ao final, as matrizes de decomposição podem ser recuperadas.

É baseada nessa estrutura de *systolic array* que foi desenvolvida a solução em GPU aqui exposta no próximo capítulo.

4 Implementação em CUDA

Este capítulo fornece uma idéia geral de como foi realizada a implementação dos algoritmos explorados no capítulo anterior no ambiente de desenvolvimento CUDA. É nele que começarei a discutir as reais contribuições que obtive durante o desenvolvimento desse trabalho de graduação. Serão discutidos os principais pontos de paralelização dos algoritmos e a razão de alguns pontos chaves terem sido escolhidos.

Cabe nesse capítulo falar um pouco da arquitetura de CUDA, de modo a introduzir certa terminologia e para se ter uma idéia de como funciona o seu modelo de programação. Assim, a primeira seção será dedicada a expor o básico sobre CUDA.

A partir daí, as seções seguintes irão focar na implementação de cada um dos algoritmos citados anteriormente. Teremos inicialmente a seção que fala sobre o RANSAC, revelando em que pontos ele foi paralelizado. Logo após, será discutida a implementação do algoritmo de *eight-point*. Então, é relatado como foi realizada a implementação do *systolic array* voltado para computação do SVD em CUDA. Por último, está a seção que fala sobre a avaliação dos modelos gerados pelo algoritmo de *eight-point*. Notar que nenhum detalhe profundo em código é exibido, visando uma simplicidade da exposição.

4.1 CUDA

CUDA é o nome dado à arquitetura para programação em paralelo de propósito geral produzida pela NVIDIA [19]. CUDA foi lançado em novembro de 2006 de modo a permitir a utilização da computação em paralelo das GPUs da NVIDIA, permitindo resolver diversos problemas computacionalmente caros de uma maneira mais eficiente do que seriam resolvidos em CPU.

O desenvolvimento de *softwares* em CUDA é baseado na amplamente conhecida linguagem de programação C, estendendo a sua sintaxe. Programadores que tem alguma experiência com essa linguagem poderão facilmente se adaptar à CUDA, apenas atentando a entender o novo modelo de computação em paralelo que é lhe dado acesso.

Para ter acesso ao ferramental provido por CUDA, há três componentes de *software* que devemos utilizar: CUDA SDK, CUDA *Toolkit* e o *driver*. O *Toolkit* é o ambiente que permite a programação de soluções na linguagem C para CUDA. Além do compilador, vem com uma

ferramenta de *profile* chamada *Profiler* e algumas bibliotecas básicas: CUDA FFT, que permite o uso da transformada de Fourier rápida e BLAS, que possui operadores de álgebra linear básicos. O SDK possui diversos programas de exemplo para facilitar e exemplificar códigos desenvolvidos para a plataforma.

4.1.1 Modelo de programação

As GPGPUs provêem uma capacidade de computação enorme a quem as utiliza. Isso ocorre devido ao fato de que elas possuem diversas unidades de processamento em paralelo. O programador em CUDA deve saber utilizar essas diferentes unidades o melhor possível, de modo a não criar (ou minimizar) dependências entre diferentes unidades de processamento. Para tal, é necessário entender o modelo de programação que CUDA oferece.

Thread, *block* (bloco) e *grid* são importantes conceitos em CUDA. São eles que abstraem a visão do programador das unidades de processamento da GPGPU. Cada *thread* corresponde a uma unidade de execução em paralelo. Um conjunto de *threads* é organizado segundo o programador em um bloco. Cada *thread* de um mesmo bloco possui acesso a uma memória compartilhada (*shared memory*) extremamente eficiente. O *grid* é um conjunto de blocos. Esses três conceitos definem a chamada hierarquia de *threads*.

Para definir funções que são executadas na GPGPU, o programador declara os chamados *kernels*. Na chamada de execução de cada *kernel*, o programador define exatamente qual a organização que ele vai querer de blocos no *grid* e de *threads* em cada *bloco*. Os blocos em um *grid* podem ser organizados até tridimensionalmente, assim como as *threads* nos blocos. Cada uma das *threads* endereçadas nessa organização possuirá um único identificador que permitirá o acesso a ela. Cada uma delas executará o mesmo *kernel* e o programador poderá diferenciá-las através de desvios condicionais baseados, por exemplo, no identificador de cada *thread*.

Além de definir a hierarquia de *threads*, é importante definir a hierarquia de memórias. As *threads* de CUDA podem acessar diferentes níveis de memória. Cada uma das *thread*s em execução possui sua própria memória local. Além disto, *thread*s de um mesmo bloco possuem acesso a uma memória compartilhada, como citado anteriormente, que pode ser acessada por todas as *thread*s deste bloco. No nível mais alto da hierarquia existe a memória global, a qual todas as *thread*s em execução têm acesso. Além da memória global, temos também a memória de textura e a memória constante, as quais todas as *thread*s também

possuem acesso. Essas são memórias apenas de leitura, que possuem diferentes propriedades que podem ser utilizadas durante a execução.

A figura abaixo resume as hierarquias tanto de memória quanto de *thread*s.

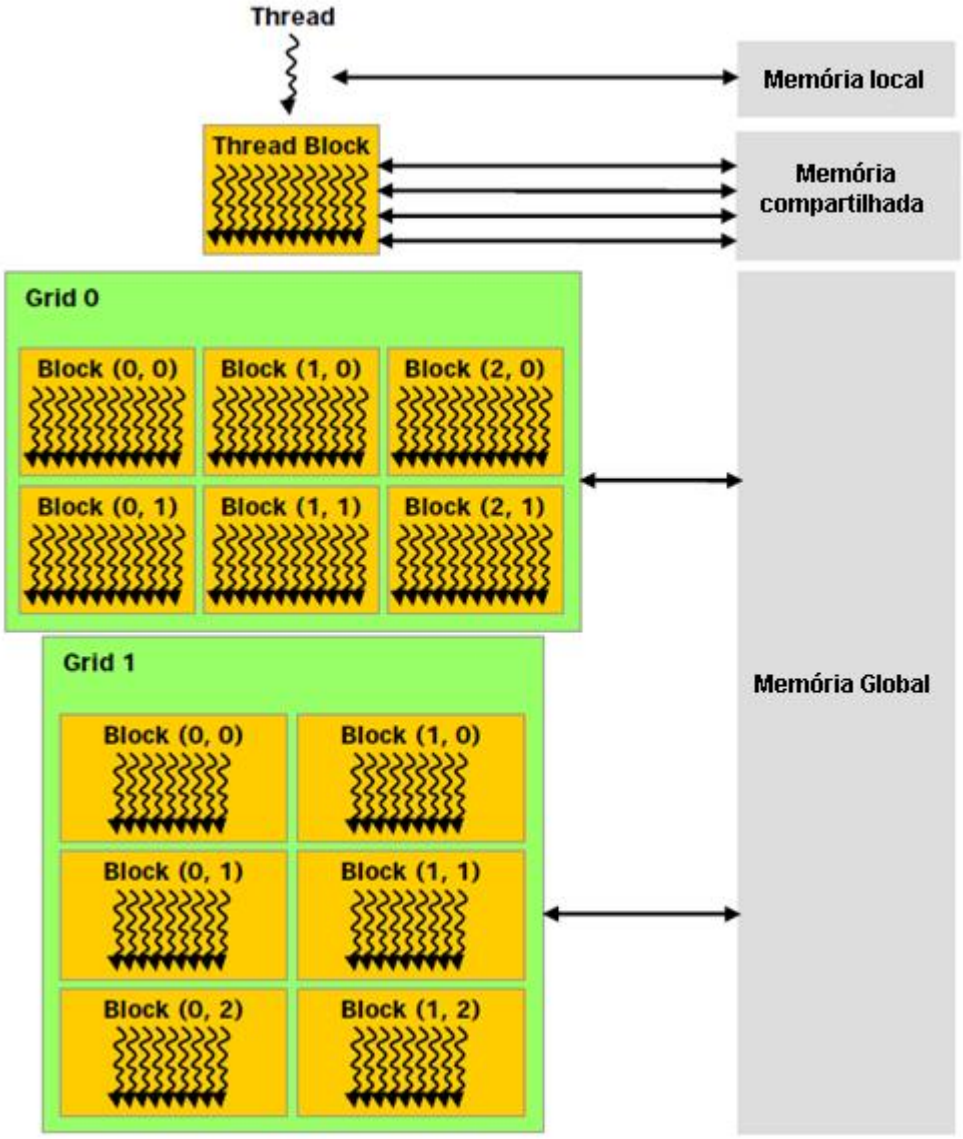

Figura 4.1 - Hierarquias de *threads* e memória de CUDA.

Não serão dados nesse texto detalhes de sintaxe relativos à extensão da linguagem C para CUDA. Para mais detalhes, pode-se consultar o *CUDA Programming Guide* [13], que constitui uma boa introdução ao funcionamento de CUDA.

4.2 Implementação do RANSAC

No capítulo anterior, pudemos ter uma idéia de como pode ser realizada a implementação do RANSAC. Aqui serão feitas algumas considerações sobre os pontos em que o RANSAC pode ser paralelizado, tentando mostrar as possibilidades mais interessantes e justificar a escolha dos pontos de paralelização pelos quais optei ao final.

Uma maneira interessante de paralelizar o RANSAC leva em conta dividi-lo em duas partes: a parte em que geramos os modelos a serem avaliados e a parte da avaliação dos modelos. O RANSAC prevê que os modelos gerados sejam avaliados logo em seguida, mas dessa maneira, o resultado da operação de computar o modelo deve ser utilizado como entrada para a etapa de avaliação do modelo, impedindo a realização em paralelo desses duas etapas. Além disto, a estrutura de computação do modelo do *eight-point* pode ser vista como totalmente diferente da estrutura de computação de avaliação de *inliers* relativa ao *eight-point*.

De modo a resolver estes problemas, foram criados dois *kernels* durante a implementação em CUDA. Cada um dos *kernels* resolve separadamente os dois problemas: o primeiro é responsável pela criação de vários modelos em paralelo e o segundo responsável pela avaliação em paralelo de cada um destes modelos.

O primeiro *kernel*, chamado de `eightpointKernel`, recebe como entrada as correspondências entre as imagens e executa várias computações de matrizes fundamentais ao mesmo tempo, executando o algoritmo de *eight-point* em paralelo diversas vezes, cada uma delas utilizando amostras mínimas selecionadas aleatoriamente. Ao final, ele obtém como resultado as várias matrizes fundamentais relativas a cada modelo computado.

É importante observar que, antes dessa primeira etapa, a computação aleatória de quais correspondências serão amostras mínimas de cada modelo é realizada previamente, em CPU. A justificativa para tal escolha é a ausência de uma função pseudo-aleatória provida pelas bibliotecas de CUDA. A amostra mínima utilizada no algoritmo de *eight-point* nesse caso é de 10 pontos, podendo ser facilmente alterada.

A seguir, o segundo *kernel*, chamado `detect`*`inliers`*`Kernel`, avalia cada um dos modelos gerados em paralelo, contando o número de *inliers* de cada um (utilizando-se da função de distância para tal). Ao final da execução deste *kernel*, teremos em mãos a

quantidade de *inliers* relativa a cada modelo gerado. Destes, o modelo que possuir mais *inliers* é escolhido como melhor modelo.

Dessa maneira, o RANSAC é dividido em várias iterações, cada uma delas executando uma determinada quantidade de modelos e os avaliando. O melhor modelo da iteração atual é comparado ao melhor modelo global obtido até agora, este último sendo substituído caso possua menos *inliers*. O número total de modelos gerados pelo algoritmo dependerá do parâmetro N, a quantidade máxima de iterações do RANSAC. A cada iteração, este parâmetro é atualizado como descrito no capítulo anterior. Este parâmetro de número de iterações pode ser ajustado para diferentes casos.

Ao final das diversas iterações e obtenção de uma matriz fundamental juntamente com os *inliers* do modelo, o RANSAC pode refinar a matriz fundamental utilizando-se de uma complementação do algoritmo em CPU, com a execução do *eight-point* com todos os *inliers*.

Repare que a maneira como o algoritmo foi implementado pode ser adaptada para qualquer problema que se deseje uma estimação robusta de um determinado modelo. Isto significa que a função aqui construída pode ser generalizada facilmente.

4.3 Implementação do *eight-point*

No último capítulo foi detalhado o funcionamento do algoritmo de *eight-point*. É interessante reparar que pouco do algoritmo envolve computações que possam ser realizadas em paralelo. Isto pode ser visto observando que a computação de cada etapa depende de resultados das etapas anteriores e, portanto as etapas do algoritmo devem ser implementadas seqüencialmente.

Apesar de não ser possível a paralelização do algoritmo como um todo, podemos paralelizar as etapas do algoritmo separadamente. Ao fazer isso, devemos atentar para as etapas que mais tomam tempo ao serem realizadas seqüencialmente. Foi justamente em função destes pontos que a paralelização foi executada.

Como dito na seção anterior, a função `eightpointKernel` é responsável pela execução do algoritmo de *eight-point*. Ela está organizada de modo que cada bloco é responsável por realizar cinco execuções do algoritmo de *eight-point* em paralelo. Cada uma dessas execuções tem à disposição 25 *thread*s que são organizadas diferentemente dependendo da etapa que está sendo executada.

Na primeira etapa, as correspondências da amostra mínima relativa a esse *eight-point* são carregadas na memória compartilhada a partir da memória de textura, onde elas foram guardadas previamente após terem sido recebidas como entrada. Durante essa etapa, é realizada a normalização das coordenadas dos pontos em cada imagem, como prevê o algoritmo de *eight-point* (normalizado). Basicamente, 20 das 25 *thread*s tratam os pontos de correspondência separadamente (10 de cada imagem, como definido na seção anterior). Esses 20 pontos após normalizados gerarão a matriz fundamental.

Após a etapa de normalização, as *thread*s são responsáveis por construir a matriz que será dada como entrada à primeira execução de SVD. Essa primeira matriz é 10×9, pois o número de pontos na amostra mínima é de 10. O SVD dessa matriz é computado dentro do próprio *kernel* do *eight-point*. Mais detalhes sobre a organização das *threads* para essa operação serão vistos na seção de SVD.

Ao final da computação, obtemos um vetor com os valores da matriz fundamental. Algumas *threads* são alocadas para estabelecer a partir desse vetor a matriz 3×3 da qual se fará a computação do segundo SVD. Após zerar o menor valor singular, a matriz é recomposta através do produto da sua decomposição.

Por fim, a matriz fundamental obtida é denormalizada para poder ser utilizada de acordo com as coordenadas originais dos pontos das imagens.

4.4 Implementação do SVD

A razão da necessidade de implementar o SVD em CUDA é o fato de não haver uma implementação padrão existente para uso durante a época do início do trabalho. Como o objetivo é implementar em CUDA o algoritmo de *eight-point*, é necessário que tenhamos acesso a um procedimento que calcule esta decomposição. Na verdade esta foi uma das implementações que mais tomou tempo e mudou o foco inicial do projeto, por se tratar de um problema mais abstrato de se lidar. Atualmente, já existem algumas implementações em CUDA do SVD, como por exemplo **Erro! Fonte de referência não encontrada.**.

Dentro do *eight-point*, a execução do SVD é realizada duas vezes, uma delas numa matriz 10×9 e outra numa matriz 3×3. No primeiro caso, todos as 25 *threads* do `eightpointKernel` serão utilizadas. No segundo caso, apenas quatro delas. Uma

importante nota é a de que todas as operações são realizadas na memória compartilhada do bloco, por sua extrema eficiência.

No SVD, as *threads* são organizadas bidimensionalmente e cada uma delas é responsável por uma submatriz 2×2, como explicado no capítulo anterior. Antes de iniciar o procedimento de decomposição, a matriz 10×9 é montada como uma matriz 10×10 cuja última coluna é preenchida com zeros, exceto pelo último elemento que é um. Assim, existem exatamente 25 submatrizes 2×2, cada uma delas sendo acessada por apenas uma *thread*, que realiza as operações de obtenção dos ângulos de rotação, caso pertençam à diagonal, ou apenas realizam a operação de rotação com os parâmetros recebidos das submatrizes pertencentes à diagonal, caso contrário. Tudo isto visando implementar a estrutura de *systollic array*, como explicado no capítulo anterior.

Neste primeiro caso, como o resultado do procedimento será apenas o último vetor da matriz V da decomposição, não precisamos manter a matriz U na memória compartilhada durante a execução, podendo economizar memória por este aspecto. Se necessitássemos também da matriz U aqui, teríamos que alocar mais 10×10 *floats* de memória.

No caso da matriz 3×3, ela é completada com zeros para formar uma matriz 4×4, exceto pelo último elemento, que é um. Temos aqui quatro submatrizes 2×2. Dessa vez, estamos interessados em todos os elementos da decomposição e por isso, temos que manter a matriz U para uso posterior. Porém, a quantidade de memória aqui utilizada é menor. Apenas necessitamos de uma matriz U de tamanho 4×4.

Inicialmente, o algoritmo do SVD foi implementado em separado, visando à execução de testes de corretude dos seus resultados. Essa implementação foi a utilizada em comparações com o algoritmo em CPU. Uma limitação do algoritmo foi constatada durante sua implementação: como ele é executado totalmente utilizando a memória compartilhada, a qual possui um limite de 16 KB, o tamanho de matrizes de entrada é limitado. Durante o capítulo de resultados, falarei mais sobre isto.

4.5 Avaliação dos modelos

Para realizar a avaliação dos modelos gerados, é interessante a criação de uma nova estrutura de *grid*. Aqui, cada bloco será responsável por um modelo a ser avaliado, de modo que cada *thread* de um bloco será responsável por verificar se um determinado ponto de

entrada é classificado como *inlier* do modelo gerado. Caso isso não seja possível, pois em CUDA há um limite de 512 *threads* por bloco, colocamos mais blocos por modelo. Assim, se o número de correspondências de entrada for maior que 512, a tarefa de verificar um único modelo é dividida por mais de um bloco.

A função de distância usada já foi definida no capítulo anterior. Cada *thread* apenas verifica se a correspondência que ela tem é ou não um *inlier* do modelo. Ao final, é contado e retornado o número de *inliers* do modelo.

5 Resultados

Este capítulo traz os principais resultados obtidos durante a evolução deste trabalho. Os principais resultados aqui apresentados serão dados através da comparação dos algoritmos implementados com uma implementação em CPU.

Ressalto que as implementações em CPU foram realizadas utilizando-se uma biblioteca auxiliar chamada VXL [14]. A VXL é uma biblioteca muito utilizada quando se trata de problemas em visão computacional. Ela provê tipos básicos pré-definidos que foram muito úteis durante a implementação em CPU, tais como vetores e matrizes. Além disto, ela tem uma função que calcula a decomposição em valores singulares de uma matriz, facilitando o trabalho.

Este capítulo inicia com uma seção sobre as implementações em CPU dos algoritmos anteriormente comentados. Logo após, a metodologia de comparações é delineada, e os cenários de teste são exibidos. Na última seção, apresento os resultados baseados nessa metodologia através de gráficos e tabelas contendo *speedups* obtidos na GPU.

5.1 Desenvolvimento na VXL

Como já dito, a VXL possui funções e tipos que propiciam maior facilidade durante o desenvolvimento das aplicações em visão computacional. A sua função que calcula a decomposição SVD foi a utilizada aqui. Apesar de possuir os algoritmos de *eight-point* e RANSAC já implementados, resolvi fazer, utilizando a biblioteca, versões análogas às implementadas em GPU, apenas usando funções de nível mais baixo da biblioteca, como as funções de tratamento de matrizes e vetores. Dois programas foram criados em C++ utilizando-se a VXL.

Um destes programas toma como entrada matrizes quaisquer e obtém ao final o SVD delas. Ele consiste simplesmente em chamadas sucessivas em seqüência à função que obtém o SVD de uma matriz de entrada. Ele foi construído de modo a dar suporte a qualquer quantidade e ordem de matriz. É importante comentar que o algoritmo de SVD da VXL é completamente diferente do implementado em CUDA. A VXL deve utilizar alguma implementação padrão de bibliotecas de álgebra linear, tal como a LINPAC [11].

O segundo programa é o RANSAC com algoritmo de *eight-point*. A seqüência implementada foi construída exatamente como descrito nos capítulos anteriores. Com as funções providas pela VXL, tal programa foi muito mais simples de se construir.

5.2 Metodologia

Durante o processo de comparação dos algoritmos implementados, tomeu como base uma comparação centrada no desempenho do tempo necessário para execução deles. Nenhum estudo sobre a quantidade de memória envolvida durante a computação dos algoritmos foi realizado. Numa comparação desse tipo a CPU obteria trivialmente melhores resultados, por a CPU computar seqüencialmente e, portanto necessita apenas dos dados que estão sendo trabalhados no momento.

Cada comparação levou em conta diferentes configurações de entrada para os algoritmos. Em todas elas, diversas execuções do mesmo cenário foram realizadas, pois uma medida de tempo de execução é um valor que pode ser considerado uma variável aleatória. Além disto, o próprio RANSAC possui a característica de algoritmo probabilístico, possuindo, portanto diferentes saídas para uma mesma entrada. Em todas as comparações, cinco repetições do mesmo experimento foram realizados. Os resultados apresentados são todos em termos de média de tempo de execução por cenário.

No algoritmo de decomposição SVD, foram realizados testes variando-se o tamanho de matrizes e também o número de matrizes que são dadas como entrada. Já no RANSAC com *eight-point*, o principal parâmetro variado durante as execuções foi o parâmetro t, que dá o limiar máximo de distância que é admitido para uma correspondência ser *inlier* do modelo. Mudando tal parâmetro, tanto se pode facilitar a execução dos algoritmos (aumentando ele, de modo a um modelo bom ser facilmente encontrado), quanto podemos aumentar a precisão da matriz fundamental obtida ao final da execução do algoritmo.

Para o algoritmo de SVD, foram gerados 90000 números aleatórios uniformemente entre zero a serem usados como entrada. Estes números foram organizados de diferentes maneiras de modo a produzir matrizes de diferentes tamanhos. No caso do RANSAC com *eight-point*, dados reais compostos por 304 correspondências entre duas imagens foram obtidas utilizando o algoritmo SIFT [18] de *feature tracking* e utilizados como entrada para o algoritmo.

Vale ressaltar que a contagem de tempo em GPU ainda inclui o tempo de transferência de memória da CPU para a GPU. Em todas as comparações esse tempo de transferência, que não inclui nenhuma computação, foi incluído. Já em CPU, esse tempo extra não existe.

Nos algoritmos em CPU, os testes foram realizados num computador Desktop com um processador core 2 duo de 2.8 GHZ e 4 GB de memória RAM. Uma GeForce 8600 GT foi utilizada como GPU de testes. Essa placa não é uma das placas topo de linha da NVIDIA, assim execuções com placas de maior desempenho levarão a resultados muito melhores que os aqui obtidos.

5.3 Comparações

Aqui se iniciam de fato as comparações dos algoritmos. Na primeira subseção, os resultados do algoritmos de SVD são discutidos. Na segunda, os do algoritmo de RANSAC com *eight-point*.

5.3.1 SVD

No primeiro tipo de comparação produzido relativo ao algoritmo de SVD, o número de matrizes de entrada é aumentado enquanto que a ordem do tamanho da matriz é mantida constante. Nessa comparação, repetiram-se cinco vezes as execuções de cada um dos algoritmos. A Tabela 5.1 – Média de tempos de execução em ms no primeiro cenário. exibe a média dos valores de tempo de execução dos algoritmos em milissegundos, para matrizes 6×6, em função do número de matrizes. Tal tamanho foi selecionado por possuir uma ordem de grandeza compatível com as das matrizes utilizadas no algoritmo de *eight-point*. Logo em seguida, apresento um gráfico comparativo.

Aqui, pode-se notar o crescimento trivialmente linear do tempo de execução em CPU com o aumento do número de matrizes. Isto se dá simplesmente pela capacidade seqüencial de execuções que a CPU realiza. Em GPU, o crescimento também é praticamente linear, como também poderíamos esperar, apenas com algumas poucas variações. A média de *speedup* obtido neste cenário foi de $4 \times$.

	100 matrizes	500 matrizes	1000 matrizes	1500 matrizes	2000 matrizes	2500 matrizes
CPU	4,49	22,54	45,65	66,87	89,42	111,13
GPU	1,56	6,93	9,44	19,97	19,94	33,23

Tabela 5.1 – Média de tempos de execução em ms no primeiro cenário.

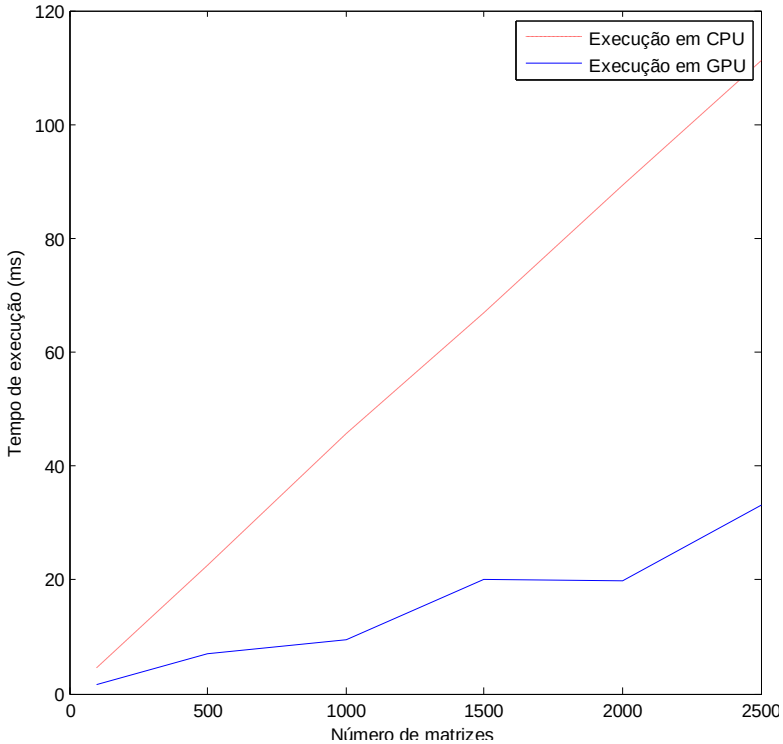

Figura 5.1 – Comparação de tempo de execução em ms do SVD com matrizes 6 × 6 como entrada.

No próximo cenário observado, o número de matrizes de entrada é mantido constante e igual a 225. Aqui, foi variado o tamanho das matrizes de entrada. Abaixo, uma tabela e gráfico das médias de tempo obtidas nestas execuções.

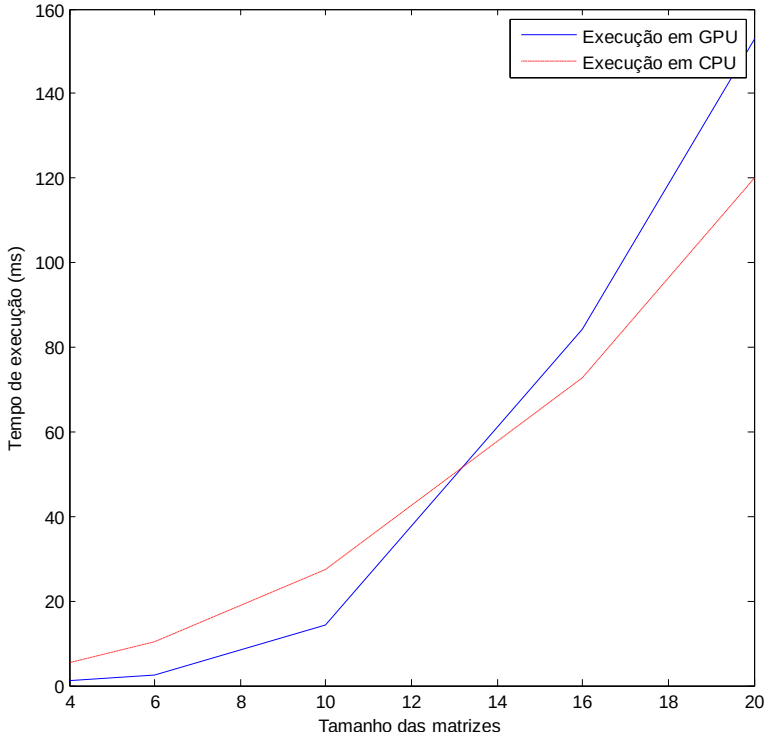

Figura 5.2 – Comparação de tempo de execução em ms do SVD com 225 matrizes de entrada.

	4x4	6x6	10x10	16x16	20x20
GPU	1,06	2,34	14,42	84,12	152,90
CPU	5,50	10,39	27,44	72,72	120,01

Tabela 5.2 – Média de tempos de execução em ms com 225 matrizes de entrada.

Um mesmo cenário foi executado com 100 matrizes de entrada, obtendo um resultado análogo a este, mostrado na tabela a seguir.

	4x4	6x6	10x10	16x16	20x20
GPU	0,57	1,56	6,11	39,09	67,52
CPU	2,70	4,49	11,66	31,70	52,48

Tabela 5.3 – Média de tempos de execução em ms com 100 matrizes de entrada.

Podemos reparar que inicialmente a curva de tempo de execução em GPU mostra um desempenho maior do que a em CPU. Com o aumento do tamanho das matrizes, a GPU começa a perder em desempenho. Isto ocorre porque a implementação do SVD em GPU é dependente da memória compartilhada que os blocos têm disponível. Como todas as operações de SVD para cada matriz ocorrem num mesmo bloco e utilizando apenas a memória compartilhada (para melhor desempenho), a quantidade de memória (16 KB por bloco) é um limitante no número de SVDs que podem ser realizados num mesmo bloco. Quanto menor o tamanho das matrizes de entrada, mais delas podem ser executadas num mesmo bloco. Quando o tamanho da matriz aumenta, apenas uma matriz é tratada num mesmo bloco, diminuindo muito o desempenho. Isto também leva a um limite no tamanho das matrizes de entrada, que foi verificado ser de 24×24.

Apesar deste problema, não há nenhuma necessidade de computar SVD de matrizes grandes, do ponto de vista do *eight-point*. O tamanho máximo das matrizes que são computados os SVDs no *eight-point* é de 10×9, como já explicado. Dessa maneira, a região que o algoritmo de *eight-point* trabalha é a região em que o desempenho em GPU ainda é mais rápido que em CPU.

Na região em que o desempenho em GPU é melhor, é observada uma média de *speedup* de $4 \times$. Para matrizes na ordem em que o *eight-point* trabalha, o *speedup* do SVD é de em média $5 \times$.

5.3.2 RANSAC com *eight-point*

Os resultados das comparações dos algoritmos de RANSAC com *eight-point* em geral foram muito interessantes. Um *speedup* de até 40 × foi obtido. Neste cenário de comparação, o limiar de distância máxima foi variado. Como explicado, limiares maiores tendem a facilitar o cálculo de um modelo, enquanto que limiares menores dificultam a geração, mas geram modelos mais precisos. Vale ressaltar que o critério de parada do algoritmo foi baseado no número de iterações calculado adaptativamente como citado em capítulos anteriores, ou no número máximo de iterações pré-definido.

O algoritmo em GPU foi executado modificando-se um parâmetro: a quantidade de *eight-point* realizada por iteração do algoritmo (isto é, por chamada de *kernel*). Diminuir este parâmetro deve fazer o algoritmo executar em um tempo menor no caso de que o limiar é alto, enquanto que o aumento dele leva a mais modelos gerados e avaliados por iteração, o que é necessário em caso de limiares baixos. Uma análise empírica pode levar a uma boa escolha desse número. As médias de tempo obtidas durante dez execuções em cada cenário é exibida na tabela e gráfico abaixo. Para facilitar, o cenário com 100 *eight-point*s por iteração é chamado de GPU – 100, enquanto que o de 300 é chamado de GPU – 300.

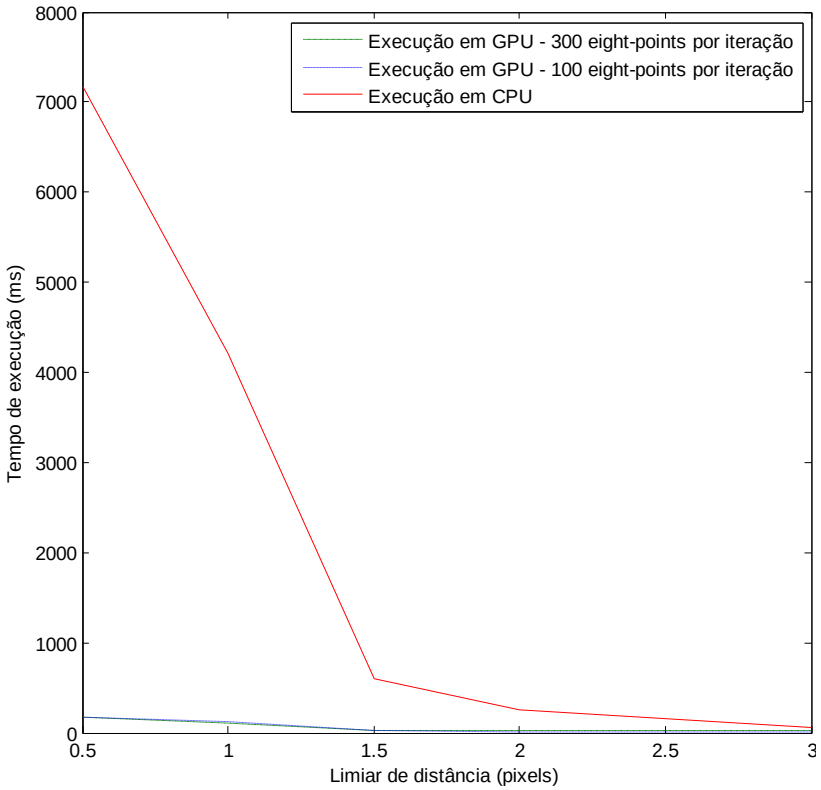

Figura 5.3 – Comparação de tempo de execução de execução em ms do RANSAC.

Limiar	0,5 pixels	1 pixels	1,5 pixels	2 pixels	3 pixels
GPU - 100	171,63	105,29	28,39	28,41	28,35
GPU - 300	172,84	118,37	19,07	9,65	9,61
CPU	7168,50	4215,60	600,39	250,32	62,27

Tabela 5.4 – Média de tempos de execução em ms, em função do limiar definido em *pixels*.

A diferença entre tempos de execução é enorme para casos com limiar baixo. Mesmo para limiares altos, um *speedup* bom foi obtido. Este é o resultado da paralelização massiva da geração de modelos. A média de *speedup* obtida é de $28\times$ com relação à execução com GPU – 100 e de $23\times$ com relação à GPU – 300. O melhor *speedup* obtido foi de $41\times$ para o caso de limiar $0{,}5$, e no caso de limiar 3, foi obtido um *speedup* de $6\times$. É interessante ressaltar que com o limiar de $0{,}5$, nenhum algoritmo conseguiu calcular um modelo com bastante *inliers* antes de o número máximo de iterações pré-definido ser atingido.

A Figura 5.4 – Comparação do RANSAC com diferentes números de *eight-points* por iteração. trás a diferença somente entre as execuções em GPU, para os dois diferentes parâmetros utilizados. Para os casos mais difíceis, há um ligeiro ganho ao utilizar-se um maior número de *eight-points* por iteração do RANSAC. Porém, há um determinado tempo mínimo para a execução do algoritmo (o tempo que leva apenas uma iteração do RANSAC) em função do número de *eight-points* por iteração. Assim, como esperado, se o limiar de distância for muito baixo, é mais interessante o uso de um número maior de *eight-points* por iteração.

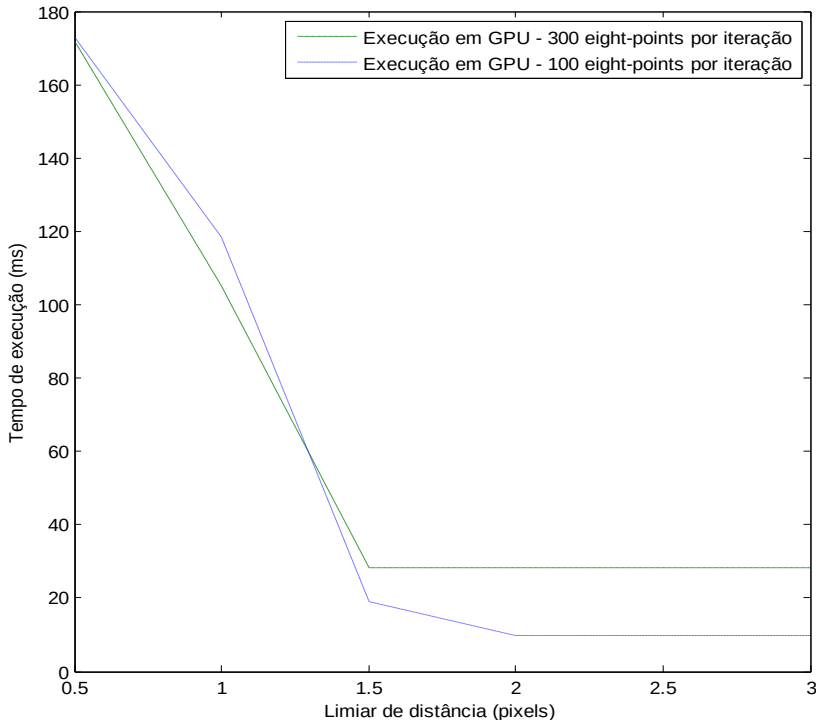

Figura 5.4 – Comparação do RANSAC com diferentes números de *eight-points* por iteração.

5.4 Considerações finais

É necessário entender um pouco sobre a precisão dos algoritmos citados. Sobre o SVD, [12] diz que com 6 a 10 varreduras do algoritmo, o resultado da operação já converge com grande precisão, no sentido de que a soma dos elementos fora da diagonal principal da matriz dos valores singulares obtida fica da ordem de 10^{-12} vezes a soma dos elementos fora da diagonal principal da matriz original. Em todas as execuções aqui apresentadas, foram realizados 10 varreduras durante a execução do algoritmo do SVD, garantindo assim a precisão citada. Outro jeito de garantir a precisão do algoritmo é exigir que ele execute diversas varreduras até atingir uma precisão pré-determinada.

Detalhes sobre como as etapas subseqüentes do *eight-point* (como a etapa de zerar o menor valor singular) alteram a precisão da estimação inicial da matriz fundamental podem ser encontrados em [4] e são inerentes a qualquer algoritmo que utilize o SVD para reduzir o posto da matriz fundamental.

A tabela abaixo resume os *speedup*s apresentados neste capítulo.

	melhor caso	média	pior caso
SVD - número de matrizes variável	4,83	3,69	2,88
SVD - tamanho de matrizes variável	5,18	2,63	0,78
RANSAC - GPU - 100	41,76	22,79	2,20
RANSAC - GPU - 300	41,47	28,20	6,48

Tabela 5.5 – Resumo dos *speedup*s obtidos.

6 Conclusão e Trabalhos Futuros

Este é o capítulo final do trabalho. Nele, tentarei apresentar um resumo das principais implicações diretas dos resultados obtidos neste trabalho de graduação. Além disto, como acho que o trabalho aqui exposto abre espaço para novas idéias sobre o assunto, farei algumas sugestões de como ele pode ser complementado. Vou falar também das restrições que o projeto possui, encontrando assim pontos do trabalho que podem ser mais bem desenvolvidos, e também encaixados como trabalhos futuros. Também serão tratados os pontos em que houve maior dificuldade durante o desenvolvimento do trabalho.

A principal dificuldade encontrada foi a inicial mudança de foco do trabalho em vista de obter uma implementação eficientemente paralelizável de um algoritmo de SVD. Esta implementação demandou um grande esforço, por ser a mais matematicamente envolvida, porém foi concluída (não por completo, como comentado abaixo) e conseguiu dar suporte ao trabalho.

Os resultados aqui obtidos foram, no geral, interessantes. O algoritmo de SVD conseguiu um *speedup* de aproximadamente $3\times$ na região de trabalho do RANSAC. Com certeza um *speedup* dessa ordem de magnitude ao ser executado diversas vezes paralelamente dentro do RANSAC contribuiu para gerar uma diferença enorme entre as execuções seqüencial e em paralelo do algoritmo. Este é provavelmente um ponto importante o qual, apesar de não ser algo de tanto destaque quando visto como um resultado independente, resultou a diferença no conjunto por completo.

Em relação ao RANSAC com *eight-point*, uma média de *speedup* de $28\times$ parece ser bastante satisfatória. Este resultado era de se esperar, devido à facilidade de se paralelizar diferentes gerações de modelos. O RANSAC gera diversos modelos independentes diversas vezes. Nada mais natural do que uma idéia deste tipo funcionar bem em um ambiente de computação paralela. Assim, a principal idéia deste trabalho de paralelização do algoritmo com ganhos em desempenho foi confirmada ser correta. Creio que este seja o resultado principal desse trabalho.

O algoritmo de decomposição de matrizes SVD aqui desenvolvido ainda não é geral o bastante. Devido a ele utilizar apenas de operações em memória compartilhada, é fácil constatar que ele é limitado pelo tamanho dessa memória. Apesar disto não influenciar nos resultados sobre o RANSAC com *eight-point*, não é natural que um algoritmo esteja apenas

funcional para alguns casos. Uma proposta de trabalho futuro então é a de ampliar o algoritmo de modo a ele sustentar os casos de matrizes com ordens maiores, atentando para a possibilidade de haver alguma perda no desempenho.

Uma maneira de se fazer isso, talvez seja tratar as trocas de memória mais custosas em memória compartilhada e as menos em memória global dando acesso por outras *threads* a uma mesma matriz. Essa solução inicial talvez não seja de bom desempenho, mas já é uma idéia que pode ser seguida em algum outro trabalho.

Outro jeito seria implementar a decomposição através de multiplicações de matrizes, as quais já possuem uma implementação padrão em CUDA, como simplesmente descrito em [8], sem a necessidade de simulação da estrutura de *systolic array*. Talvez, o uso de algum outro método paralelizável diferente do aqui exposto pode ser tentado também.

Neste trabalho, a questão precisão do resultado dos algoritmos não foi observada, por se tratar de um ponto sobre o qual já existem diversos estudos na literatura. Mais detalhes sobre um estudo desse tipo sobre o *eight-point* podem ser encontrados em [3] e [4], que comentam inclusive sobre a normalização das coordenadas dos dados e as conseqüências da normalização. Ainda assim, outro trabalho futuro poderia ser uma comparação em termos de precisão $\times$ desempenho com outros algoritmos (ou conjunção de algoritmos) que tentem estimar a matriz fundamental.

7 Referências

[1] R. Hartley and A. Zisserman, Multiple View Geometry in Computer Vision. Cambridge Univ. Press, 2000.

[2] H.C. Longuet-Higgins, “A Computer Algorithm for Reconstructing a Scene from Two Projections,” Nature, vol. 293, no. 10, pp. 133-135, Sept. 1981.

[3] R. Hartley, “In Defense of the Eight-Point Algorithm,” IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 19, no. 6, pp. 580-593, June 1997.

[4] W. Chojnacki, M.J. Brooks, A. van den Hengel, D. Gawley, “Revisiting Hartley's normalised eight-point algorithm,” IEEE Trans. Pattern Analysis Machine Intelligence, 25, 9, 2003, 1172-1177.

[5] Martin A. Fischler and Robert C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". Comm. of the ACM 24: 381–395.

[6] Gerald E. Farin, “NURBS: From Projective Geometry to Practical Use,” A. K. Peters, Ltd., Natick, MA, 1999.

[7] M. Pollefeys, “Obtaining 3D Models with a Hand-Held Camera/3D Modeling from Images,” Course/Tutorial notes, presented at Siggraph 2002/2001/2000, 3DIM 2001/2003, ECCV 2000. http://www.cs.unc.edu/~marc/tutorial/

[8] Weiwei Ma, M. E. Kaye, D. M. Luke, R. Doraiswami, “An FPGA-Based Singular Value Decomposition Processor,” Canadian Conference on Electrical and Computer Engineering, 2006. CCECE '06, May 2006 Page(s):1047 - 1050.

[9] R. P. Brent, F. T. Luk, C. Van Loan, “Computation of the Singular value Decomposition Using Mesh-Connected Processors,” Journal of VLSI and Computer Systems, vol. 1, no. 3, pp. 242-270, 1985.

[10] G. H. Golub and F. T. Luk, "Singular Value Decomposition: applications and computations," Trans. 22nd Conf. Army Mathematicians, ARO Report 77-1, pp. 577-605, 1977.

[11] J. J. Dongarra, C.B. Moler, J.R. Bunch, G.W Stewart, “LINPAC User's Guide” SIAM, Philadelphia, Pa., 1979.

[12] G.H Golub, C. Reinsch, “Singular value Decomposition and least squares solutions,” Numer. Math. 14, 1971.

[13] Compute Unified Device Architecture. URL: http://www.nvidia.com/cuda.

[14] VXL. URL: http://vxl.sourceforge.net/.

[15] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Eurographics 2005*, 2005, pp. 21-51.

[16] David Luebke , Greg Humphreys, How GPUs Work, Computer, v.40 n.2, p.96-100, February 2007.

[17] Yi Ma , Stefano Soatto , Jana Kosecka , S. Shankar Sastry, An Invitation to 3-D Vision: From Images to Geometric Models, SpringerVerlag, 2003.

[18] David G. Lowe. "Distinctive image features from scale-invariant keypoints." International Journal of Computer Vision, 60(2):91–110, 2004.

[19] NVIDIA. URL: http://www.nvidia.com.

[20] Jie Ming, Huang Xianlin, "A Lunar Terrain Reconstruction Method Using Long Base-line Stereo Vision," Control Conference, 2007. CCC 2007. Chinese , vol., no., pp.488-492, July 26 2007-June 31 2007.

[21] Mehta, B.V., Marinescu, R., "Comparison of image generation and processing techniques for 3D reconstruction of the human skull," Engineering in Medicine and Biology Society, 2001. Proceedings of the 23rd Annual International Conference of the IEEE , vol.4, no., pp. 3687-3690 vol.4, 2001.

[22] Kirli, Y., Ulusoy, I., "3D reconstruction of underwater scenes from uncalibrated video sequences," Signal Processing and Communications Applications Conference, 2009. SIU 2009. IEEE 17th , vol., no., pp.105-108, 9-11 April 2009.

[23] Feng Zhang, Limin Shi, Zhenhui Xu, Zhanyi Hu, "A 3D Reconstruction System of Indoor Scenes with Rotating Platform," Computer Science and Computational Technology, 2008. ISCSCT '08. International Symposium on , vol.2, no., pp.554-558, 20-22 Dec. 2008.

[24] Thiago Farias, João Marcelo Teixeira, Pedro Leite, Gabriel Almeida, Veronica teichrieb, Judith Kelner, "High performance computing: CUDA as a supporting technology for next generation augmented reality applications", Brazilian Symposium on Computer Graphics and Image Processing, Campo Grande, 2008.

[25] Sheetal Lahabar, P. J. Narayanan, "Singular value decomposition on GPU using CUDA," ipdps, pp.1-10, 2009 IEEE International Symposium on Parallel&Distributed Processing, 2009.