UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

GUSTAVO ILHA

MPVUE - PLATAFORMA MULTIPROCESSADOR EM CHIP PARA VISÃO COMPUTACIONAL

Porto Alegre
2019

GUSTAVO ILHA

MPVUE - PLATAFORMA MULTIPROCESSADOR EM CHIP PARA VISÃO COMPUTACIONAL

Tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Doutor em Engenharia Elétrica.

Área de concentração: Engenharia de Computação

ORIENTADOR: Prof. Dr. Altamiro Amadeu Susin

Porto Alegre
2019

GUSTAVO ILHA

MPVUE - PLATAFORMA MULTIPROCESSADOR EM CHIP PARA VISÃO COMPUTACIONAL

Esta tese foi julgada adequada para a obtenção do título de Doutor em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Altamiro Amadeu Susin, UFRGS
Doutor pela Polytechnique National Institute of Grenoble, France

Banca Examinadora:

Prof. Dr. Fernando Santos Osório, USP São Carlos
Doutor pela Institut National Polytechique de Grenoble – Grenoble, França

Prof. Dr. Sérgio Bampi, UFRGS
Doutor pela Stanford University – Stanford, Estados Unidos

Prof. Dr. Gilson Inácio Wirth, UFRGS
Doutor pela Universitaet Dortmund – Dortmund, Alemanha

Prof. Dr. Edison Pignaton de Freitas, UFRGS
Doutor pela Universidade Federal do Rio Grande do Sul – Porto Alegre, Brasil

Prof. Dr. Ivan Müller, UFRGS
Doutor pela Universidade Federal do Rio Grande do Sul – Porto Alegre, Brasil

Coordenador do PPGEE: _____
Prof. Dr. João Manoel Gomes da Silva Jr.

Porto Alegre, agosto de 2019.

DEDICATÓRIA

Ao meu avô, Arno Müller, pela inspiração.

AGRADECIMENTOS

À minha família. À minha esposa Manuela, incansável e afável. Aos meus pais, Nelson e Adriane, pelo exemplo de servir. Aos meus irmãos Felipe e Mariana, pelo apoio. Aos meus avós, Arno, Lourdes, Henrique e Walmy, pelo exemplo de simplicidade e retidão.

Ao time do MPVue, Cezar Reinbrecht, Anderson Silva, Rafael Viana, Luis Filipe Meyer, Douglas Lawish, Davi Lazzaroto, Eduardo Costa e Vitor Junior por acreditar em uma ideia o suficiente para transformá-la em realidade.

Aos colegas de LaPSI, Fabio Pereira, Joel Luft, Thiago Waszak, Marcelo Negreiros e Ronaldo Husemann pelo convívio e as muitas ideias trocadas.

Ao professor Altamiro Amadeu Susin, pelos conselhos, orientações e principalmente pelos questionamentos engrandecedores.

À Universidade Federal do Rio Grande do Sul, por ser um farol de conhecimento em tempos de tempestade do saber.

RESUMO

O processamento de imagens e a visão computacional evoluíram significativamente nos últimos anos com o progresso da microeletrônica e dos sensores de imagem. A visão é muito útil para os animais se movimentarem e interagirem com o meio ambiente. Para os seres humanos, é ainda mais importante, pois a maioria das atividades depende da capacidade de ver e entender o contexto visual. Imagens estão em toda parte em nossas vidas diárias para comunicação, saúde, transporte e muitos outros aspectos. A visão é uma função muito complexa: além da aquisição de imagens, muitas outras tarefas são necessárias para compor um modelo de cena. Por enquanto, estamos interessados na captura de informações visuais para construir um modelo de cena 3D e reconhecer sinais e formas, tentando imitar a capacidade natural de ver. O sistema pode ser embarcado em objetos móveis ou pontos de observação, como veículos autônomos, sistemas de assistência ao motorista (DAS), monitoramento de tráfego e vigilância. Esta tese apresenta a plataforma MPVue, um Sistema MultiProcessador Heterogêneo de Memória Distribuída em Chip (DM-HMPSoC) estruturado sobre uma rede em chip (NoC) em malha 2-D, adaptada para realizar eficientemente o paralelismo de comunicação e execução em sistemas embarcados, visando aplicações de visão computacional. A arquitetura é descrita em uma RTL sintetizável e validada em FPGA. A comparação de desempenho foi feita usando algoritmos tipicamente usados no Processamento de Imagens: um Filtro Passa Baixas e a FFT. Uma implementação de arquitetura de software flexível baseada na Arquitetura Orientada a Serviços (SOA) facilita o encadeamento de funções para diferentes aplicativos. A arquitetura MPVue é adequada para os processos de visão de alto nível e trabalhos futuros avaliarão, por exemplo, seu desempenho para executar uma CNN treinada para capturar imagens de placas de veículos em vídeos de tráfego. Estudos em andamento definirão e portarão um sistema operacional para aumentar o desempenho do agendamento de tarefas e para executar vários aplicativos simultaneamente. Além disso, a API deve ser padronizada para facilitar o desenvolvimento de novos aplicativos.

Palavras-chave: Processamento de Imagens, Visão Computacional, Estrutura a partir do Movimento, SLAM Monocular, SW/HW codesign, FPGA, Rede em Chip, Sistemas em Chip Multiprocessados.

ABSTRACT

Image processing and computer vision had evolved significantly in recent years hanged on the progress of microelectronics and image sensors. Vision is very helpful for animals to move and interact with the environment. For human beings, it is even more important since most activities rely on the ability to see and understand the visual context. Images are everywhere in our daily lives for communication, health-care, transportation and many other domains. Vision is a very complex function indeed: in addition to the image acquisition many other tasks are needed to compose a scene model. For now, we are interested in the capture of visual information to build a 3D scene model and recognize signs and shapes, trying to mimic the natural ability to see. The system may be embedded on moving objects or observation points like autonomous vehicles, Driver Assistance Systems (DAS), traffic monitoring and surveillance. This thesis presents the MPVue platform, a Distributed Memory Heterogeneous MultiProcessor System on Chip (DM-HMPSoC) structured over a 2-D Mesh Network on Chip (NoC), tailored to efficiently perform communication and execution parallelism on embedded systems, targeting computer vision applications. The architecture is described in a synthesizable RTL and was validated on a Xilinx KC705 development board. To check the flexibility and the performance of the architecture, two versions were generated: one with 3X3 NoC and one with a 4X4 NoC. Performance comparison was done using algorithms typically used on Image Processing: a Low Pass Filter and Fast Fourier Transform. A flexible software architecture implementation based on Service-Oriented Architecture (SOA) facilitates functions chaining for different applications. The MPVue architecture is suitable for the high-level vision processes and future works will evaluate, for example, its performance to run a convolutional neural network (CNN) trained to grab vehicle license images on traffic video. Ongoing studies will define and port an OS to increase task scheduling performance and to run multiple applications simultaneously. Furthermore, the API must be standardized ease the development of new applications.

Keywords: Image Processing, Computer Vision, Structure from Motion, Monocular SLAM, SW/HW codesign, FPGA, Network-on-Chip, Multiprocessor System-On-Chip.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AMBA	*Advanced Microcontroller Bus Architecture*
AXI	*Advanced eXtensible Interface*
BA	*Bundle Adjustment*
BNN	*Binarized Neural Network*
BRAM	*Block RAM*
BRIEF	*Binary Robust Independent Elementary Features*
CISC	*Complex-Instruction-Set Computers*
CMOS	*Complementary Metal-Oxide Semiconductor*
CNN	*Convolutional Neural Network*
CPU	*Central Processing Unit*
DIT	*Decimation-in-Time*
DNA	*Deoxyribonucleic acid*
DPA	*Differential Power Analysis*
DSP	*Digital Signal Processing*
ECCV	*European Conference on Computer Vision*
EKF	*Extended Kalman Filter*
FF	*Flip-Flop*
FFT	*Fast Fourier Transform*
FPGA	*Field-Programmable Gate Array*
FPS	Frames por Segundo
FPU	*Floating-point Unit*
GCC	*GNU Compiler Collection*
GPIO	*General Purpose Input Output*
GPS	*Global Positioning System*
HW	*Hardware*
IDE	*Integrated Development Environment*

IoT	*Internet of Things*
IP	*Intellectual Property*
ISA	*Instruction Set Architecture*
JPEG	*Joint Photographic Experts Group*
LaPSI	Laboratório de Processamento de Sinais e Imagens
LED	*Light Emitter Diode*
LiDAR	*Light Detection and Ranging*
LUT	*Look-Up Table*
MJPEG	*Motion Joint Photographic Experts Group*
MPSoC	*Multi Processor System-on-Chip*
NoC	*Network-on-Chip*
OpenCV	*Open source Computer Vision library*
ORBIS	*OpenRisc Basic Instruction Set*
OS	*Operating System*
PC	*Personal Computer*
PLL	*Phase-locked Loop*
PnP	*Perspective-n-Point*
PPGEE	Programa de Pos-Graduação em Engenharia Elétrica
PULP	*Parallel processing Ultra Low power Platform*
PTAM	*Parallel Tracking and Mapping*
RAM	*Random Access Memory*
RANSAC	*Random Sample Consensus*
RISC	*Reduced-Instruction-Set Computer*
3D-RMS	*3D Reconstruction Meets Semantics*
ROM	*Read-Only Memory*
RTL	*Register-Transfer Level*
SfM	*Structure from Motion*
SIFT	*Scale-Invariant Feature Transform*
SLAM	*Simultaneos Location and Mapping*
SURF	*Speeded-Up Robust Features*
SoC	*System-on-Chip*
SW	*Software*
UART	*Universal Asynchronous Receiver/Transmitter*
UFRGS	Universidade Federal do Rio Grande do Sul

ULA Unidade Lógica e Aritmética

VLIW *Very-Long-Instruction Word*

SUMÁRIO

1 INTRODUÇÃO

Sistemas de visão biológicos são excelentes em extrair e analisar as informações essenciais para as funções vitais, como navegar por ambientes complexos, achar comida ou escapar de um perigo. Essas tarefas são realizadas com grande sensibilidade e alta confiabilidade, dado que imagens naturais são muito ruidosas, variáveis e ambíguas (MEDATHATI *et al.*, 2016). O sistema de visão possui como características: ser passivo – não precisa interferir no ambiente para realizar a medição –, direcional e medir um fenômeno físico extremamente abundante em nosso ambiente: a presença de luz e suas reflexões, com uma riqueza de detalhes impressionante.

Até alguns anos atrás, o tema de processamento de imagens estava restrito a aplicações em ambientes extremamente controlados e com condições de contorno muito bem definidas. As câmeras eram caras e os algoritmos necessários para as funções mais básicas necessitavam de poder computacional não disponível para as aplicações-alvo.

Com a popularização de equipamentos capazes de gerar, armazenar e transmitir vídeos, o tema de processamento de imagens possui um novo papel na sociedade. A grande quantidade de dados, aliada à rapidez exigida para que as novas informações sejam geradas e distribuídas, abre caminho para o processamento automático desses dados. As informações extraídas das sequências de vídeos são utilizadas das mais variadas formas: geração de estatísticas, classificação, reconhecimento de pessoas ou objetos, e, mais recentemente, para a localização do observador.

Entretanto, o desempenho obtido pelos mais poderosos dispositivos, para reconhecer o ambiente e os objetos que compõem a cena em que estão inseridos ainda não podem ser comparados com as soluções encontradas na natureza. Por isso, novos métodos devem ser buscados de forma a aproximar as soluções tecnológicas das biológicas.

1.1 Motivação

Robôs móveis devem conseguir se localizar e se mover em ambientes complexos, bem como interagir com pessoas e objetos. O mesmo ocorre com os veículos autônomos, que devem identificar a localização exata de outros veículos e pedestres no contexto em

que estão inseridos. Para resolver tais problemas, a utilização de câmeras pode ser a solução, ou até mesmo o fator decisivo para permitir o desenvolvimento de respostas para estes desafios. (NAWAF; TRÉMEAU, 2014; ZHAO *et al.*, 2013; ENGEL; STURM; CREMERS, 2013).

Por outro lado, sistemas de mapeamento devem identificar e localizar as silhuetas do local a ser mapeado, bem como dos objetos que nele se situam (FIELDS *et al.*, 2009). De forma similar, os sistemas de monitoramento não mais são suficientes se apenas gravarem as imagens de um ambiente: é necessário que as informações relevantes sejam extraídas em tempo real. Além disso, movimentações precisam ser entendidas como normais ou não, pessoas ou objetos não pertencentes ao lugar precisam ser identificadas e situações anômalas precisam ser detectadas o quanto antes.

Buscando solucionar todas essas demandas, algumas respostas têm apresentado grande complexidade, outras buscam a execução em tempo real, através da simplificação dos algoritmos (ENGEL; SCHöPS; CREMERS, 2014), ou utilizando uma implementação que busca certo nível de paralelismo (geralmente multi-thread) (SONG; CHANDRAKER; GUEST, 2016). Embora muitos algoritmos sejam implementados para execução sequencial em PC, a própria natureza de suas tarefas permite que sejam projetados para execução em paralelo, em núcleos de processador separados, em pequenos núcleos dedicados àquela atividade, que, agrupados, geram a resposta final.

Essa linha de raciocínio pode ser utilizada para os algoritmos clássicos do processamento de imagens – dos mais simples aos mais complexos. Estes podem ser transformados em rotinas paralelas e autônomas, tais como a segmentação, onde todos os pixels devem ser comparados com um valor padrão e decidido a que grupo pertencem. Ao invés de varrer todos os pixels, pode-se criar comparadores locais que realizam a operação e entregam apenas o resultado final (BORENSTEIN; ULLMAN, 2008; BINH; LOI; THUY, 2012; ZHANG; JI, 2010). Por outro lado, o filtro de partículas, onde cada partícula pode ser representada como um núcleo preditor/corretor completo, independente dos outros, deve ser realimentado a cada iteração independente dos demais (ZHAO *et al.*, 2013; CENTIR *et al.*, 2013; WANG; CHANG; CHEN, 2009).

A implementação em hardware sempre andou junto com o processamento de imagens, principalmente mais recentemente. (BRENOT; PIAT; FILLATREAU, 2016) explora a implementação em hardware buscando um melhor desempenho do sistema como um todo. Entretanto, as soluções apresentadas ainda não atendem plenamente à demanda de um sistema completo e integrado capaz realizar a tarefa de forma autônoma.

1.2 Objetivos

O objetivo da presente tese é desenvolver uma plataforma de software e de hardware que possibilite o desenvolvimento de sistemas embarcados capazes de, em tempo

real, identificar o ambiente à sua volta, permitindo, a partir da localização do observador, construir uma nuvem de pontos de onde seja possível extrair as informações relevantes para seu objetivo, seja de simples deslocamento evitando choques, seja a identificação da presença de objetos e suas distâncias.

No contexto da presente tese, o conceito de tempo real é compreendido como a execução do algoritmo proposto dentro do tempo de amostragem da sequência de imagens de entrada.

1.3 Contribuições

Durante o desenvolvimento da plataforma, diversas soluções foram desenvolvidas e testadas. De forma a contribuir com o estado da arte, busca-se otimizar os métodos de reconstrução 3D desde o hardware em que ele opera até o software que implementa de fato a funcionalidade. Portanto, essa tese apresenta as seguintes contribuições ao estado da arte:

- Uma metodologia para o cálculo da profundidade é demonstrada, em conjunto com uma estimativa de erro baseado no angulo do efeito de paralaxe;
- Uma análise das estimativas de erro do valor calculado para a profundidade, em conjunto com uma análise da efetividade do uso de limiares no resultado final;
- Um algoritmo para estimar a estrutura 3D da cena a partir de múltiplas câmeras embarcadas em um robô submetido ao 3D-RMS (TYLECEK *et al.*, 2019);
- Uma arquitetura de hardware eficiente, utilizando uma NoC de RISC-V para paralelizar soluções de visão computacional [1];
- Uma arquitetura de software embarcado baseada em programação orientada a serviços com execução eficiente que gera menos desperdício de tempo de computação e maior flexibilidade para a aplicação [2];

Por fim, a construção de um hardware capaz de acelerar as tarefas de alta demanda computacional, seja por otimização dos recursos, seja por uso de paralelismo, aliado a uma exploração desse potencial pelo software em todas as camadas de abstração torna toda a solução mais eficiente.

[1] ILHA, G.et al.MPVue – A Multi-Processor System-on-Chip Platform forComputer Vision. submetido à publicação. 2019

[2] ILHA, G.et al.MPVue – A Multi-Processor System-on-Chip Platform forComputer Vision. submetido à publicação. 2019

1.4 Organização

Buscando contribuir com o estado da arte em processamento de imagens, a presente tese é organizada da seguinte forma: o Capítulo 2 introduz os fundamentos dos sistema de projeção 3D-2D contido nas câmeras e apresenta uma revisão dos algoritmos de visão computacional, fechando com as caraterísticas típicas dos sistemas multiprocessados utilizados nesse tipo de solução. O Capítulo 3 faz a revisão da literatura sobre Visão Computacional, trazendo os algoritmos básicos para a área, bem como os trabalhos relevantes para a área de reconstrução 3D. Além disso, um estudo sobre as plataformas de hardware disponíveis para o desenvolvimento da solução é apresentado. O Capítulo 4 descreve as plataformas utilizadas para o desenvolvimento do presente trabalho. O Capítulo 5 detalha a solução proposta para reconstrução 3D, situando-a no contexto das soluções já existentes. O Capítulo 6 apresenta o passo a passo da implementação do protótipo do algoritmo proposto, bem como seus resultados, contendo análise de erro e medição de desempenho. O Capítulo 7 detalha a proposta do sistema em hardware para processamento de imagens embarcado. O Capítulo 8 descreve a arquitetura de software orientada a serviços. O Capítulo 9 expõe os detalhes da implementação em hardware, e apresenta discussão sobre os resultados obtidos da plataforma. O Capítulo 10 mostra o estudo de caso, onde uma aplicação de reconstrução 3D foi desenvolvida sobre a plataforma MPVue, utilizando os algoritmos do estado da arte. Por fim, o Capítulo 11 termina por apresentar as conclusões.

2 FUNDAMENTAÇÃO TEÓRICA

Nesse Capítulo é demonstrado o embasamento da física e matemática utilizada no processamento de imagens, principalmente no que tange à geometria da cena e à interação causada pelo movimento da câmera com o ambiente. Em seguida são mostrados os algoritmos de processamento de imagens necessários para a solução dos problemas de reconstrução 3D. Por fim, o embasamento teórico relativo a implementação de MPSoCs é apresentado, mostrando a classificação de arquitetura interna no que tange a organização da memória e os tipos de conjuntos de instruções.

2.1 Projeção de Perspectiva em 3D

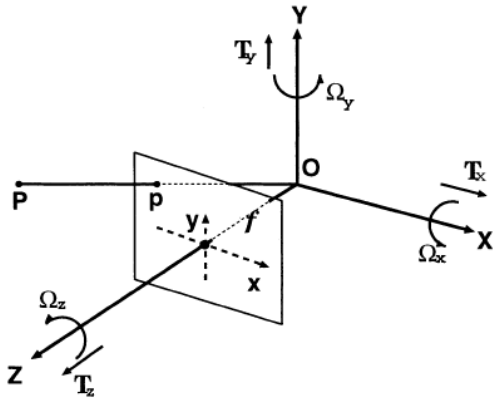
O modelo utilizado para mapear os pontos no espaço para uma imagem bidimensional em uma câmera é chamado de modelo *pinhole*. Esse modelo é baseado nas leis da geometria óptica. Apesar de existirem outros modelos, como lentes olho-de-peixe ou omnidirecionais, o modelo *pinhole* será utilizado. Nesse modelo, a lente da câmera é representada pelo seu centro óptico. Esse ponto está situado entre a cena tridimensional e o plano bidimensional de imagem. Além disso, o eixo óptico, que é perpendicular ao plano de imagem, passa pelo centro óptico. O ponto de intersecção entre o eixo óptico e o plano de imagem é chamado de ponto principal. A distância entre o ponto principal e o centro óptico é chamado de b. Para lentes reais, a distância b é sempre maior que a distância focal f da lente. Contudo, o valor b pode ser aproximado por f se a distância entre o objeto e a lente for muito maior que b (WÖHLER, 2013). Nesse modelo, a imagem é formada invertida no plano de imagem passando pelo centro óptico.

2.1.1 Sistemas de coordenadas

A partir da definição do modelo de câmera, é possível definir o sistema de coordenadas relativo à câmera, chamado C. A Figura 1 mostra a proposta de sistema de coordenadas (X, Y, Z) do observador (O) e o correspondente sistema de coordenadas da imagem (x, y), localizado no plano perpendicular ao eixo Z e distante f da origem (O). A fim de facilitar a representação, o plano de imagem é virtualmente colocado em frente ao ponto

principal. Nesse sistema, o ponto P - localizado no espaço 3D - é projetado no ponto p no sistema de coordenadas da imagem, formado por um plano. Para tanto, a câmera captura a imagem da cena desprezando a profundidade. Os objetos são projetados no plano da imagem descartando a informação de distância no eixo Z. Isso também implica que pontos que estão alinhados no espaço entre si e o centro óptico são projetados no plano da imagem em um único ponto.

Figura 1 – Sistema de coordenadas (X, Y, Z) centralizado no observador (O) e o sistema de coordenadas da imagem (x, y)



Fonte: (YANG; OE; TERADA, 2001)

Considera-se que o eixo óptico da lente da câmera esteja alinhado com o eixo Z de seu sistema de coordenadas. Para obter as coordenadas no plano da imagem de um ponto no espaço 3D, primeiramente se traduz o ponto para o sistema de coordenadas da câmera – centralizado no ponto focal da lente. Após, é feita a projeção do ponto no plano da imagem.

Para isso, consideram-se (x, y) as coordenadas projetadas no plano da imagem e (X, Y, Z) as coordenadas do ponto no espaço 3D. Consideram-se, também (X_0, Y_0, Z_0) como sendo as coordenadas da câmera no espaço 3D.

Dessa forma, segundo (HEEGER; JEPSON, 1992) a projeção dos pontos do espaço 3D no plano da imagem pode ser definida como:

$$x = f\frac{X - X_0}{Z - Z_0}, \qquad y = f\frac{Y - Y_0}{Z - Z_0} \tag{1}$$

As coordenadas (x, y) no plano da imagem são medidas na mesma unidade que (X, Y, Z) e b. O ponto central é definido como $x = y = 0$.

2.1.2 Parâmetros Intrínsecos da Câmera

O processo de formação da imagem depende, também, de parâmetros intrínsecos da câmera. Eles são dependentes da montagem da câmera, de sua lente e do sensor utilizado para amostrar o sinal. Utilizando o modelo *pinhole*, os parâmetros são descritos como a

distância principal b, o tamanho de pixels k_y e k_x nos eixos vertical e horizontal, respectivamente, o ângulo θ do plano de imagem e as coordenadas do ponto principal (x_0, y_0). Para as câmeras modernas, $\theta = 90°$ e $k_y = k_x$ (WÖHLER, 2013).

Além disso, deve ser levada em conta a distorção da lente, que causa resultados indesejados nas coordenadas dos pontos da cena projetados no plano da imagem. Tanto os parâmetros intrínsecos quanto a distorção da câmera podem ser estimados durante o processo de calibração da câmera. O processo de correção das imagens é bastante estudado (WÖHLER, 2013). Nesse trabalho, as imagens serão utilizadas sempre após o processo de compensação da distorção. Os parâmetros intrínsecos são considerados dados de entrada e serão chamados de matriz K, contendo o ponto focal em relação ao eixo x e ao eixo y $f = (f_x, f_y)$ e o ponto central $c = (c_x, c_y)$. Portanto, uma forma alternativa para reescrever de forma matricial a conversão do espaço 3D para o sistema de coordenadas do plano da imagem é

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_c \tag{2}$$

2.1.3 Conversão de Sistemas de Coordenadas

Considerando um sistema de coordenadas arbitrário r, a transformação de um ponto representado nesse sistema $P_r = [X, Y, Z]_r^T$ para outro sistema de coordenadas s é representado por uma matriz de transformação $M_{r,s}$. Essa matriz é composta por uma parte rotacional $R_{r,s}$, que corresponde a uma matriz 3x3 ortonormal, determinada por três parâmetros independentes – os ângulos de rotação de Euler $\Omega_{r,s}$ – e uma parte translacional $T_{r,s}$, representando a diferença entre os sistema de coordenadas. Assim, o cálculo das coordenadas do ponto (P_s) no sistema s pode ser representada como

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_s = \begin{bmatrix} R_{11_{r,s}} & R_{12_{r,s}} & R_{13_{r,s}} \\ R_{21_{r,s}} & R_{22_{r,s}} & R_{23_{r,s}} \\ R_{31_{r,s}} & R_{32_{r,s}} & R_{33_{r,s}} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_r + \begin{bmatrix} T_{x_{r,s}} \\ T_{y_{r,s}} \\ T_{z_{r,s}} \end{bmatrix} \tag{3}$$

Desse modo, a transformação de um sistema de coordenadas global W para o sistema de coordenadas da câmera C é representado por uma matriz $M_{W,C}$, composto por uma matriz de rotação $R_{W,C}$ e um vetor de translação $T_{W,C}$. De forma análoga, a conversão de um ponto do sistema de coordenadas da câmera para um sistema de coordenadas global pode ser feito de forma inversa, utilizando uma matriz $M_{C,W}$.

Um movimento da câmera pode ser representado pela mesma formulação, como se fosse uma transformação de coordenadas para um novo espaço. Dessa forma, um movi-

mento completo da câmera pode ser representado por $\vec{\Theta} = (\vec{T}, \vec{\Omega})$, onde a translação é $\vec{T} = (T_x, T_y, T_z)$ e a rotação é $\vec{\Omega} = (\Omega_x, \Omega_y, \Omega_z)$. Por fim, a Equação (3) pode ser utilizada para transpor os pontos entre sistemas de coordenadas de uma câmera que se move ao longo do tempo.

2.1.4 Fluxo Óptico

Conforme demonstra (HEEGER; JEPSON, 1992), todos os pontos de um corpo rígido e estático compartilham os mesmos parâmetros de movimento que o sistema de coordenadas da câmera. Devido ao movimento da câmera, o movimento relativo de um ponto no espaço é dado por

$$\begin{bmatrix} \frac{\partial X}{\partial t} \\ \frac{\partial Y}{\partial t} \\ \frac{\partial Z}{\partial t} \end{bmatrix} = -\left(\begin{bmatrix} \Omega_x \\ \Omega_y \\ \Omega_z \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_c + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \right) \tag{4}$$

Derivando-se a Equação (1) em relação ao tempo e substituindo na Equação (4) se obtém

$$\vec{\Delta}(x, y) = \rho(x, y)A(x, y)\vec{T} + B(x, y)\vec{\Omega} \tag{5}$$

onde $\rho(x, y) = \frac{1}{Z}$ é a profundidade inversa, e onde:

$$A(x, y) = \begin{bmatrix} -f & 0 & x \\ 0 & -f & y \end{bmatrix} \tag{6}$$

$$B(x, y) = \begin{bmatrix} \frac{xy}{f} & -(f + \frac{x^2}{f}) & y \\ f + \frac{y^2}{f} & \frac{xy}{f} & -x \end{bmatrix} \tag{7}$$

A Equação (5) representa o fluxo óptico como função do movimento 3D e da profundidade (HEEGER; JEPSON, 1992).

É importante notar que $A(x, y)$ e $B(x, y)$ dependem apenas dos dados disponíveis na imagem e de nenhum dado desconhecido. Portanto, a Equação (5) é considerada bilinear; $\vec{\Delta}$ é linear para $\vec{T}$ e $\vec{\Omega}$ quando $\rho(x, y)$ for fixado, e é linear para $\rho(x, y)$ e $\vec{\Omega}$ quando $\vec{T}$ for fixado.

2.1.5 Cálculo das Coordenadas 3D

Se os valores de $\rho(x, y)$ são conhecidos, é possível inferir a posição do ponto no espaço de coordenadas da câmera. Assim, considerando que as coordenadas do observador

$(X_0, Y_0, Z_0) = (0, 0, 0)$, pode-se isolar X e Y, respectivamente, na Equação (1) e obter:

$$X = \frac{x}{f\rho(x,y)} \qquad Y = \frac{y}{f\rho(x,y)} \qquad Z = \frac{1}{\rho(x,y)} \tag{8}$$

Com essa Equação, pode-se calcular os pontos ($P_c = (X, Y, Z)_c$) em relação ao sistema de coordenadas da câmera.

2.1.6 Erro de Reprojeção

O trabalho de (TRIGGS, 1996) discute diferentes aspectos, incluindo o modelo de projeção e a parametrização do problema de reconstrução 3D. Para definir o erro de predição de um ponto – também conhecido como erro de reprojeção – (TRIGGS, 1996) considera um modelo simples de cena, contendo pontos 3D estáticos e individuais capturados por câmeras. Nesse modelo, esses pontos são medidos com ruído em uma imagem que os contenha. É assumido, então, que essa medida obedece a um modelo preditivo que pode ser estabelecido. Ou seja, é possível calcular, a partir da posição do ponto no espaço e a posição/orientação da câmera, qual será a posição esperada $(\bar{x}, \bar{y})$ do ponto na imagem resultante da cena, conforme a Equação (1). O erro de reprojeção é, então, definido como a diferença entre este valor e o valor medido:

$$\begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} = \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix} - \begin{bmatrix} x \\ y \end{bmatrix} \tag{9}$$

Essa Equação pode ser utilizada para medir o erro entre o valor estimado de um ponto na imagem e o efetivamente medido.

2.2 Visão Computacional

Nessa seção serão apresentadas as técnicas que são base para o estudo de Visão Computacional, mais especificamente da Estrutura a partir do Movimento (SfM). Primeiramente, serão apresentados os fundamentos básicos de processamento de imagens utilizados nessa área. Em seguida, a Seção 2.2.3 abordará as técnicas de SfM clássicas, seguidas pelas de SLAM, na Seção 2.2.4.

Do ponto de vista de processamento de imagens, para ser possível reconstruir um ambiente é necessária a estimativa da profundidade dos pontos da imagem. Isso pode ser feito utilizando uma sequência de imagens em que seus pontos – alguns ou todos – possam ser claramente identificados sequencialmente. Esses pontos escolhidos para serem identificados entre imagens chamam-se *features*. Uma *feature* é composta pela informação de sua localização e outras informações necessárias para identificá-la em outras imagens. A diferença entre o ângulo de cada *feature* entre as imagens é chamado de paralaxe, e isso é que permite o cálculo da profundidade (CIVERA; DAVISON; MONTIEL, 2008).

2.2.1 Escolha e Localização de Features

As *features*, que serão utilizadas para os cálculos de diferenças de deslocamento no plano de imagem, podem ser escolhidas de forma aleatória ou utilizando algum critério que facilite sua identificação entre imagens.

O detector de bordas de Canny (CANNY, 1987) pode ser utilizado para escolha de *features* com base na busca por bordas. Para tanto, as bordas da imagem são detectadas e a imagem é, então, subdividida em blocos. O ponto de borda mais próximo do centro do bloco é escolhido como *feature*.

Um dos algoritmos mais difundidos para escolha de *features* é o algoritmo de Shi-Tomasi (SHI; TOMASI, 1994). Esse algoritmo, baseado no método de Harris (HARRIS; STEPHENS, 1988), seleciona regiões cujos autovalores possuam valores maiores que um mínimo, quando calculados em ambas direções (horizontal e vertical). Esses valores altos indicam que a região possui alta derivada horizontal e vertical – sendo pertencente a uma borda, ou textura marcante – e, portanto, é fácil de ser rastreada. Os pontos centrais dessas regiões são escolhidos como a posição dessas *features*. Recentemente, uma versão do algoritmo de Shi-Tomasi para implementação em FPGA foi proposto por (AGUILAR-GONZÁLEZ; ARIAS-ESTRADA; BERRY, 2018). A implementação consegue executar em tempo real, com 44 *frames* por segundo, o processamento de imagens em Full HD.

O método de SIFT (*Scale-Invariant Feature Transform*) (LOWE, 2004) pode ser utilizado para geração de *features* utilizando a parte de detecção de pontos-chave. Eles são encontrados em um processo onde a imagem, em diversas escalas, é passada por um filtro gaussiano e são extraídas suas diferenças entre si – as diferenças de gaussianas. Os pontos de mínimos e máximos dessas imagens são escolhidas como *features*. Esse método foi evoluído para o método SURF (*Speeded-Up Robust Features*) (BAY; TUYTELAARS; VAN GOOL, 2006), que utiliza o operador Hessian-Laplace. Ele detecta fortes derivadas em duas direções ortogonais.

As *features* são o ponto de partida dos algoritmos esparsos e semi-densos de fluxo óptico e de reconstrução 3D, portanto, uma boa escolha do algoritmo de seleção das mesmas significa um resultado final mais robusto. Para mais informações a respeito dos métodos de escolha de *features*, o trabalho de (NOURANI-VATANI; BORGES; ROBERTS, 2012) apresenta uma comparação entre os métodos de detecção, mostrando a eficiência de diversos métodos para uso no cálculo de fluxo óptico. O trabalho conclui que os métodos estudados apresentam resultados satisfatórios, enquanto o método de Shi-Tomasi possui os melhores resultados para o caso do fluxo óptico.

2.2.2 Rastreamento de Features

Uma das etapas necessária nos algoritmos de SfM é o rastreamento de *features* entre as imagens. Dentre as soluções propostas, se destaca a solução proposta por (LUCAS; KANADE, 1981) e aperfeiçoada por (BOUGUET, 2000).

O método de cálculo de fluxo óptico é baseado na Equação (10). Nela, a intensidade do pixel é representada como uma função da posição (x, y) e o tempo t em um vídeo.

$$f(x, y, t) = f(x + u, y + v, t + \tau) \tag{10}$$

Sendo u, v e τ a variação em x, y e t, respectivamente.

Usando a aproximação de primeira ordem da série de Taylor, é possível reescrever a Equação (10) como

$$f(x, y, t) = f(x, y, t) + I_x u + I_y v + I_t \tau \tag{11}$$

sendo I_x, I_y e I_t as derivadas parciais da imagem em relação a x, y e t, respectivamente. Essa Equação pode, então, ser reescrita como

$$I_x u + I_y v = -I_t \tau \tag{12}$$

Onde as derivadas parciais I_x, I_y e I_t podem ser calculadas por subtração com o pixel adjacente nas imagens, ou outro método de cálculo.

Considerando que entre dois *frames* adjacentes a diferença de tempo seja unitária ($\tau = 1$), a Equação (12) possui duas variáveis, que não podem ser calculadas sem que outras condições sejam assumidas. Dessa forma, o método de Lucas-Kanade (LUCAS; KANADE, 1981) assume que os pontos adjacentes possuem o mesmo fluxo. Portanto, a equação pode ser resolvida por mínimos quadrados dentro de uma janela em volta do ponto desejado.

Dessa forma, segundo (BOUGUET, 2000), tem-se que

$$G = \sum_{x=p_x-\omega_x}^{x=p_x+\omega_x} \sum_{y=p_y-\omega_y}^{y=p_y+\omega_y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \qquad \bar{b} = \sum_{x=p_x-\omega_x}^{x=p_x+\omega_x} \sum_{y=p_y-\omega_y}^{y=p_y+\omega_y} \begin{bmatrix} I_t I_x \\ I_t I_y \end{bmatrix} \tag{13}$$

e o fluxo óptico ótimo $\bar{\nu}_{opt} = [u, v]^T_{opt}$ pode ser obtido por

$$\bar{\nu}_{opt} = G^{-1} \bar{b} \tag{14}$$

Essa forma somente é válida se o movimento do pixel é pequeno, de maneira que a série de Taylor permaneça válida.

Além de demonstrar as equações do método de Lucas-Kanade, a proposta de (BOUGUET, 2000) ainda sugeriu melhorias na execução do algoritmo, inserindo uma iteração no formato Newton-Raphson. Ademais, o algoritmo é executado primeiramente em versões da imagem de baixa resolução, para evitar mínimos locais e acelerar a convergência. Por fim, um esquema de cálculo de valores de intensidade sub-pixel é proposto para au-

mentar a precisão da solução.

Como consequência, é possível estimar a posição de uma *feature* entre sucessivas imagens.

2.2.3 Estrutura a Partir do Movimento

Recuperar a estrutura tridimensional de uma cena a partir de imagens é um objetivo fundamental da visão computacional (ÖZYEŞIL *et al.*, 2017). A reconstrução 3D de uma cena estacionária, utilizando diversas imagens, é tema de grande interesse em estudos de visão computacional.

Um dos primeiros trabalhos em SfM foi o de (LONGUET-HIGGINS, 1981), que introduziu um método linear de correspondência de pontos – chamado de algoritmo de oito-pontos – para a solução de SfM para duas câmeras. Esse trabalho tinha como objetivo estimar o movimento da câmera, ou seja, a rotação e a translação relativa, e as coordenadas 3D dos pontos contidos na cena, apenas utilizando dados contidos nas próprias imagens.

Outro trabalho importante foi o de (TOMASI; KANADE, 1992), que utilizou um método para estimar a posição de objetos relativamente distantes da câmera. Essa restrição permitiu considerar as projeções dos objetos no plano da câmera como paralelas, e, consequentemente, ignorar a translação da câmera na direção do eixo óptico.

A reconstrução 3D possui um amplo conjunto de soluções para as mais variadas condições de obtenção das imagens. Os métodos podem ser incrementais (WESTOBY *et al.*, 2012; WU, 2013) ou globais (CHIUSO; BROCKETT; SOATTO, 2000; MOULON; MONASSE; MARLET, 2013), de estruturas complexas (WESTOBY *et al.*, 2012) ou estimativa do movimento (PEREIRA *et al.*, 2017; CIVERA *et al.*, 2010), ou ainda métodos especificamente endereçando problemas de SfM de grande escala (CRANDALL *et al.*, 2011; ENGEL; STüCKLER; CREMERS, 2015). Tipicamente, o número de pontos em uma estrutura 3D estimada por um método de SfM é muito maior que o número de imagens utilizadas como entrada para o algoritmo. Por conseguinte, o desafio é realizar o processamento eficiente dos dados de entrada de forma consistente, mantendo a robustez ao ruído.

2.2.4 Localização e Mapeamento Simultâneo

Nesse trabalho, são tratados os casos de reconstrução de um ambiente enquanto o observador se locomove dentro desse próprio ambiente. Esse problema específico, apesar de se tratar de reconstrução 3D, é estudado como uma parte das técnicas de Localização e Mapeamento Simultâneo, pois ambas soluções compartilham as características e as restrições específicas para esse tipo de problema.

A técnica de SLAM (*Simultaneos Location and Mapping*) nada mais é que a solução para um problema comum: a exploração espacial. Consiste em mapear o ambiente em

que se encontra enquanto simultaneamente utiliza esses mapas para se localizar dentro desse ambiente (MILFORD; WYETH, 2012). Essa técnica é bastante comum em robótica móvel, carros autônomos e drones. O SLAM é efetivamente um caso especial do SfM, específico para robótica e realidade aumentada, que carrega as mesmas dificuldades do SfM (ÖZYEŞIL *et al.*, 2017), enquanto apresenta suas próprias restrições, tornando-se uma área desafiadora.

Para solucionar esse problema, a mais variada gama de sensores pode ser utilizada: sonares, radares, lasers. Todavia, independente do sensor, a forma de solução para esse problema é basicamente a mesma (HUANG; MOURIKIS; ROUMELIOTIS, 2009). O LiDAR, por exemplo, é um tipo de sensor ativo, que através de um ou mais feixes de laser é capaz de inferir a profundidade em tempo real. Apesar de tornar o problema de cálculo de profundidade trivial, o LiDAR é um sensor caro e de baixa resolução, além de sofrer degradação de performance em condições de tempo severas, como chuva e neblina (ÖZYEŞIL *et al.*, 2017). Mesmo assim, o LiDAR é muito utilizado em navegação autônoma. Os GPS diferenciais, por outro lado, são muito precisos e podem melhorar a localização, porém, são equipamentos muito caros. Ultimamente, o uso de câmeras como sensores tem se difundido muito (STRASDAT; MONTIEL; DAVISON, 2010; DAVISON *et al.*, 2007; EADE; DRUMMOND, 2006), eis que são baratas e comuns. No entanto, o processamento de imagens de alta resolução e os problemas inerentes à formação da imagem na câmera – como oclusão, ambiguidades, abertura, entre outros – tornam o uso de câmeras, por si só, um problema computacionalmente complexo para realizar o SLAM.

O trabalho realizado por (ÖZYEŞIL *et al.*, 2017) examina o problema de SfM, mostrando que esta é uma subárea da visão computacional bem desenvolvida. Entretanto, é mostrado que o posicionamento da câmera afeta a performance dos algoritmos de SfM e SLAM. O autor cita ainda que o posicionamento é especialmente crítico em sistemas de veículos autônomos, pois a câmera se move essencialmente em linha reta por longos períodos de tempo. Nesse caso específico há diversos problemas: (i) As *features* que contêm a maior quantidade de informação relevante – aquelas que possuem o maior paralaxe – estão na periferia e logo saem do campo de visão. (ii) Existem muitos mínimos locais na minimização do erro de reprojeção, que é base para muitos algoritmos. (iii) A profundidade do pixel onde não há deslocamento por paralaxe – por exemplo, o pixel central quando se anda paralelo ao eixo óptico – não pode ser obtida. Esse fenômeno induz a vários Alternativamente, algumas soluções tentam transpor esses problemas. Por exemplo, o trabalho de (VEDALDI; GUIDI; SOATTO, 2007) mostra que forçando um limite para a profundidade é possível tornar o erro de reprojeção contínuo, entretanto a solução é muito custosa computacionalmente.

Os estudos em SfM podem ser divididos em duas categorias, conforme (ÖZYEŞIL *et al.*, 2017): métodos baseados em *features* e métodos diretos. Os métodos baseados em *features* utilizam um mapa esparso de pontos da imagem e sua eficácia depende na quan-

tidade e na qualidade do conjunto de *features*. Como exemplos desses métodos pode-se citar (CIVERA; DAVISON; MONTIEL, 2008; BOTERO-GALEANO; GONZALEZ; DEVY, 2012; LIM; LIM; KIM, 2014). Por outro lado, os métodos diretos buscam utilizar a informação da imagem inteira para otimizar a posição da câmera e a estrutura 3D diretamente. O método proposto por (ENGEL; SCHöPS; CREMERS, 2014), chamado LSD-SLAM, propõe um método para construir mapas de uma cena 3D em tempo real. O mapa da cena é representado como uma coleção de *frames*-chave com restrições de posicionamento impostas por transformações de similaridade. Apesar desses métodos serem muito interessantes para reconstruir o mapa da imagem, eles tendem a ser suscetíveis à *outliers*, principalmente se a cena possui objetos não estáticos.

Outro problema para as soluções existentes, citado por (ÖZYEŞIL *et al.*, 2017), é lidar com diferentes escalas. Sensores que fornecem medição de escala, como LiDAR e câmeras estéreo, possuem um alcance limitado. Por exemplo, o cálculo de profundidade em câmeras estéreo dependem da resolução da imagem e da distância entre câmeras. O aumento do limite de resolução de profundidade passa por um ajuste físico no sistema (mudar o posicionamento das câmeras) ou por aumentar o custo computacional (tendo de processar imagens de maior resolução). Por outro lado, SLAM monocular é invariante à escala. Logo, sistemas baseados nessa solução são imunes a limitação de cálculo da profundidade relativa. Não obstante, devido ao fato de a escala não poder ser obtida diretamente, suas estimativas estão sujeitas a derivar ao longo do tempo.

Normalmente, o problema de SLAM é dividido em duas partes. A primeira busca a determinação do movimento da câmera, enquanto a segunda tem o objetivo de mapear o ambiente ao redor. A determinação do movimento da câmera atraiu um grande interesse de pesquisa que passou a se chamar de Odometria Visual. Quando a Odometria Visual é a prioridade do sistema, o mapeamento se limita a uma quantidade mínima de *features* necessária para se manter a qualidade do cálculo da posição da câmera. Nesses sistemas, o mapeamento de um ponto é normalmente desprezado assim que ele não contribui para o cálculo. Apesar disso, tanto a Odometria Visual, quanto o mapeamento do ambiente são baseados no equacionamento da geometria do espaço 3D e a forma que esse espaço é projetado no plano focal da câmera. Ademais, as principais técnicas para solucionar ambos os problemas são os mesmos. Assim sendo, alguns conceitos de Odometria Visual são expostos, de forma a serem utilizados se necessários.

2.2.5 Odometria Visual

A estimativa do movimento da câmera pode ser realizada minimizando o erro de reprojeção em algoritmos baseados em Perspective-n-Point (PnP). Segundo (PEREIRA, 2018), possuindo-se uma câmera calibrada e conhecendo pontos no espaço 3D, Perspective-n-Point é estimativa da posição e orientação da câmera que minimiza a diferença entre a posição dos pontos observados no plano da imagem e a projeção calculada para esses

mesmos pontos a partir de sua posição no espaço 3D.

Nesses sistemas, o processamento é feito *frame* a *frame* de forma intercalada. Inicialmente, o movimento da câmera é determinado baseado na estimativa existente dos pontos de interesse, comparando os valores calculados e observados da posição relativa desses pontos na imagem. Finalmente, a distância desses pontos à câmera é refinada utilizando triangulação.

Os principais algoritmos para refinamento conjunto da estrutura 3D e do movimento da câmera são conhecidos como métodos de Bundle Adjustment.

Para se reconstruir a cena, a medida do erro de reprojeção é minimizada, e o Bundle Adjustment é responsável pelo refinamento da otimização e necessita de um conjunto de estimativas para os parâmetros de câmera e da estrutura. Além disso, (TRIGGS, 1996) mostra que uma métrica estatística como essa, que permita a existência de *outliers*, é apropriada como escolha da função de custo.

2.3 MPSoC

Conforme visto até aqui, as aplicações de visão computacional necessitam uma grande quantidade de processamento para atender os requisitos de tempo de execução. Sua complexidade e desempenho demandam novas soluções para os desafios introduzidos. Assim, para ser possível embarcar as soluções, sem abrir mão do desempenho, novas arquiteturas de hardware devem ser desenvolvidas.

Os MPSoCs, Sistemas-em-Chip Multiprocessados, buscam aliar o poder computacional de vários elementos processadores usando uma interconexão rápida e confiável entre eles, como, por exemplo, uma rede em chip (NoC). Os elementos processadores podem ser simplesmente um único processador, como (TOPCUOGLU; HARIRI; WU, 2002; DOGAN *et al.*, 2012; DAVIDSON *et al.*, 2018), ou um *cluster* mais complexo que contenha mais de um processador em um único endereço da rede em chip, como (SIEVERS *et al.*, 2015; CONTI *et al.*, 2014; DINECHIN *et al.*, 2013; RAHIMI *et al.*, 2011), ou, ainda, um conjunto completamente heterogêneo de elementos processadores como processadores, aceleradores em hardware, ou outros IPs agrupados no mesmo sistema, como (WANG *et al.*, 2017; SAVAS, 2017).

Os Sistemas em Chip (SoC) são formados por diferentes componentes de um computador integrados em um único chip (HEENE *et al.*, 1989; HUNT; ROWSON, 1996; TRABER *et al.*, 2016). Quando a pesquisa de processadores migrou de um único para múltiplos *cores*, a pesquisa de SoCs começou a integrar múltiplos *cores*, de forma análoga, buscando mais desempenho (DAS *et al.*, 2018). Como as demandas por desempenho continuaram crescendo, o único processador do sistema, mesmo com múltiplos *cores*, se tornou o gargalo de desempenho. A atenção, então, se voltou para combinar mais de um processador, em conjunto com os outros subsistemas de hardware. A essa plataforma se

deu o nome de Sistemas em Chip Multi-Processados (MPSoC).

Nas últimas décadas muitas soluções utilizando MPSoCs foram propostas (JOVEN *et al.*, 2008; DOGAN *et al.*, 2012; MARONGIU; BURGIO; BENINI, 2011; SIEVERS *et al.*, 2015; ELMOHR *et al.*, 2018; KHAMIS *et al.*, 2018), tornando tal plataforma uma importante classe de sistema que incorpora os componentes necessários para uma aplicação usando múltiplos processadores programáveis como componentes (WOLF; JERRAYA; MARTIN, 2008). Aplicações em que MPSoCs são utilizados não são sistemas simples que implementam um algoritmo, mas são sistemas complexos que usam múltiplos algoritmos para resolver um problema.

Com um grande número de CPUs disponíveis no chip, maior poder computacional está disponível. Entretanto, conectar esses elementos processadores de um MPSoC se tornou um desafio relevante. Algumas tecnologias propuseram utilização de barramentos compartilhados, como Quickpath Interconnect da Intel (ZIAKAS *et al.*, 2010), ou AMBA da ARM (Arm Ltd, 1999).

O aumento de elementos processadores também aumenta o atraso de comunicação em tais barramentos, tornando-os o fator limitador de crescimento de desempenho do sistema (DAS *et al.*, 2018). A fim de buscar uma solução para essa demanda, foi proposto um novo modelo de comunicação para sistemas multiprocessados baseado em conceitos de rede, chamado rede em chip (NoC) (BENINI; MICHELI, 2002).

2.3.1 Classificação de MPSoCs

No que tange à arquitetura, sistemas multiprocessados podem ser classificados com base em sua arquitetura de memória e tipo de processador (DAS *et al.*, 2018).

A arquitetura de memória é dividida em três tipos: memória compartilhada, memória distribuída, e memória distribuída compartilhada. Sistemas de memória compartilhada basicamente adotam uma grande memória compartilhada entre todos os elementos processadores com espaço de endereçamento global. Tais elementos processadores não possuem memória local, ou quando a possuem, o fazem na forma de uma pequena cache. Esse modelo é apropriado para aplicações nas quais os dados devem poder ser acessados por qualquer um dos processadores a todo momento. Nesse modelo, há relevante latência para acessar o dado, não importando o endereço acessado. Até recentemente, essa era a organização predominante em MPSoCs (DINECHIN *et al.*, 2013; SCHWAMBACH *et al.*, 2014; ROSSI *et al.*, 2014; WÄCHTER; BIAZI; MORAES, 2011).

Por outro lado, sistemas de memória distribuída implementam cada elemento processador com sua própria memória. Cada um acessa apenas a sua memória local, com latência baixa, e se necessita dados que não estão disponíveis localmente, uma comunicação com outro elemento deve ser realizada, com um atraso. Esse modelo é apropriado quando existe uma boa relação entre a intensidade computacional e a taxa de comunicação dos dados. Muitos autores, como (MORENO *et al.*, 2008; FERNANDEZ-ALONSO *et al.*,

2010; REINBRECHT *et al.*, 2016; WANG *et al.*, 2017; CASTRILLON *et al.*, 2018), utilizam essa organização.

Sistemas de memória distribuída compartilhada implementam uma abstração no seu espaço de endereçamento. Cada elemento processador possui sua própria memória, mas o sistema implementa um tradutor para mapear cada memória local nos endereçamentos dos outros processadores, criando um espaço virtual de endereçamento compartilhado. Dessa forma, qualquer processador tem acesso a todas as memórias do sistema (MONCHIERO *et al.*, 2007; TIAN; HAMMAMI, 2009). Esse modelo aumenta a latência de acesso para memórias não-locais e pode criar um gargalo se uma parte dos dados necessita ser acessada constantemente, sobrecarregando o elemento processador onde essa parte está armazenada.

Os sistemas podem ser classificados conforme seus tipos de processadores em duas categorias: homogêneos e heterogêneos. Nos sistemas homogêneos todos os elementos processadores são exatamente do mesmo tipo. Isso simplifica a replicação, melhorando, portanto, a escalabilidade. Os estudos desenvolvidos por (WÄCHTER; BIAZI; MORAES, 2011; SIEVERS *et al.*, 2015) apresentam propostas de sistemas homogêneos. Já os sistemas heterogêneos incluem diferentes tipos de *cores* no mesmo SoC. Os estudos desenvolvidos por (ROSSI *et al.*, 2014; SAVAS, 2017; CASTRILLON *et al.*, 2018) apresentam propostas de sistemas heterogêneos. Além dos já citados, alguns sistemas heterogêneos integram lógica reconfigurável em conjunto com os elementos processadores. Esses sistemas, chamados de sistemas reconfiguráveis, permitem uma maior customização mesmo após o lançamento. Exemplos de soluções que utilizam essa técnica incluem (AYDI *et al.*, 2007; DAVIDSON *et al.*, 2018).

2.3.2 Conjunto de Instruções

Em complemento à arquitetura do sistema, para desenvolver um MPSoC, a arquitetura interna do processador também deve ser levada em consideração. Por isso, o conjunto de instruções (ISA) é um importante requisito ao selecionar o processador. A ISA é uma abstração do modelo computacional que define o comportamento do processador. Sua implementação pode variar dependendo dos limites do hardware alvo: limites de custo, área e desempenho deve ser considerados. Uma aplicação pode ser executada nas diferentes implementações de processadores que compartilham a mesma ISA, se seus limites de memória e de arquitetura forem respeitados.

As arquiteturas podem ser divididas em dois tipos: CISC e RISC. CISC são processadores com instruções grandes e complexas, que podem executar diversas operações de baixo nível. Durante o desenvolvimento dos processadores CISC, assim como os micro-programas, responsáveis por controlar as operações, cresceram, também cresceram as chances de *bugs* ocorrerem nesses programas (PATTERSON, 2018). Além disso, (EMER; CLARK, 1984) concluiu que 20% das instruções do VAX, um processador do

tipo CISC, era responsável por 60% do micro-programa, entretanto essas instruções ocupavam apenas 0,2% do tempo de execução.

Uma alternativa então foi implementar uma ISA que fosse tão simples que não necessitasse um interpretador de micro-instruções, chamada RISC. Esse formato também facilita a implementação de pipeline, o que permite taxas de *clock* mais altas (PATTERSON, 2018).

Diversas arquiteturas foram propostas, dentre as quais, destacam-se algumas. OpenRISC é uma família de processadores embarcados. A sua primeira implementação é o OpenRISC 1200, que contém um *pipeline* de cinco estágios e implementa o conjunto de instruções ORBIS (WÄCHTER, 2011). Ele pode ser implementado em arquitetura de 32 ou 64 bits do tipo Harvard, contendo cache de instrução separada da cache de dados. Ademais, possui a capacidade de expandir o conjunto de instruções utilizando componentes, como uma FPU. As aplicações-alvo deste modelo são dispositivos de telecomunicações, entretenimento doméstico e aplicações automotivas (TONG; ANDERSON; KHALID, 2006).

Outra plataforma alternativa, diferente da RISC ou CISC, é o uso de processadores VLIW. Nessa arquitetura, uma instrução pode conter múltiplas operações. O hardware, então, é compatível com essa execução em paralelo. Em vez de instruções de 32 ou 64 bits dos processadores RISC, uma instrução VLIW pode ter tamanho superior a 100 bits (PATTERSON, 2018). Em VLIW puro, não há travas entre instruções. Se o resultado de uma instrução é entrada para o cálculo de outra instrução, é responsabilidade do compilador que cuide para que a instrução subsequente rode somente quando o resultado da instrução anterior estiver disponível. Isso demanda um trabalho extra dos compiladores e dificulta a programação em assembly.

Mais recentemente, alguns processadores propostos foram baseados em uma ISA aberta. RISC-V é um exemplo de solução de ISA de código aberto (WATERMAN *et al.*, 2011) desenvolvido para dar suporte à pesquisa de arquitetura computacional. Ele é distribuído sem taxa de licenciamento para aplicações comerciais. Essa ISA é extensível por definição e muitas extensões estão disponíveis, como instrução de multiplicação, instruções de operações atômicas, instrução única para dados múltiplos (SIMD) e instruções de ponto flutuante.

2.4 Conclusão do Capítulo

Nesse Capítulo foi apresentado o embasamento da física e matemática utilizada no processamento de imagens, principalmente no que tange à geometria da cena e à interação causada pelo movimento da câmera com o ambiente. Ademais, os algoritmos de processamento de imagens que podem ser utilizados em uma solução de reconstrução 3D foram mostrados. Por fim, as características de inerentes aos sistemas multiprocessa-

dos embarcados foram listadas de forma a facilitar a classificação e comparação entre o sistema proposto e os encontrados na literatura.

3 REVISÃO DA LITERATURA

Nesse Capítulo são apresentados primeiramente alguns trabalhos relevantes para a área de SfM, que serão utilizados como base para a proposta de solução para reconstrução 3D. Depois, os trabalhos relevantes para a área de MPSoCs que serão utilizados como base de comparação para avaliar o sistema em hardware proposto são descritos.

3.1 Reconstrução 3D

(HEEGER; JEPSON, 1992) propõe um método para estimar conjuntamente a profundidade da cena e o movimento de translação e rotação da câmera. Apesar de não ser o primeiro trabalho na área, ele propõe uma nova forma de escrever a equação do fluxo óptico, separando as informações de translação e rotação em uma única equação matricial bilinear. Dessa forma, a profundidade é obtida de forma inversa, sendo linear na equação proposta. Muitos trabalhos seguintes se basearam nessa solução para tentar resolver os problemas de complexidade computacional e a presença de *outliers*. Dentre esses pode-se citar (SRINIVASAN, 2000; YANG; OE; TERADA, 2001).

Os estudos para determinar a posição da câmera, a partir de poucos pontos bem definidos na imagem são igualmente antigos. O trabalho de (HORAUD *et al.*, 1989) formalizou o problema de Perspective-n-Point (PnP) de forma objetiva. É mostrada a quantidade mínima de pontos para cada solução analítica de cálculo de posição da câmera, resolvendo o problema quando há quatro pontos em comum nas imagens. Os estudos seguindo essa abordagem evoluíram muito nos últimos anos e são a base para diversos algoritmos, principalmente de odometria visual (QUAN; LAN, 1999; GAO *et al.*, 2003; LEPETIT; MORENO-NOGUER; FUA, 2008; PEREIRA *et al.*, 2017). Essa abordagem implica em alto custo computacional para determinar a contribuição de cada *feature* no cálculo da posição da câmera. Entretanto, como necessita de muito poucas *features* para obter uma precisão adequada, seu custo total acaba sendo menor que os métodos baseados em fluxo óptico.

No trabalho iniciado por (MONTIEL; CIVERA; DAVISON, 2006) e depois evoluído por (CIVERA; DAVISON; MONTIEL, 2008), é proposto um método de parametrização

de *features* em SLAM monocular para tratar incertezas durante a inicialização dessas *features* dentro de um filtro de Kalman estendido (EKF). Para tanto, é proposto o uso do inverso da profundidade de uma *feature*, relativa à localização da câmera em que a *feature* foi vista pela primeira vez, o que produz equações de medida com um alto grau de linearidade. Dessa forma, *features* distantes podem ser utilizadas como referência para estimativas de rotação, sem a necessidade de um tratamento específico para esse caso. O trabalho trouxe resultados interessantes, mas a formulação matemática proposta apresentou grande complexidade computacional, possibilitando que apenas um número baixo de *features* fosse rastreada em tempo real. Mesmo assim, esse trabalho abriu espaço para o estudo de método de estimativa de profundidade utilizando o inverso da distância e a linearidade da sua medição, no âmbito das técnicas de SLAM sequenciais e probabilísticas. No mesmo grupo, (DAVISON *et al.*, 2007) buscou a obtenção do mapa esparso de *features* através de uma câmera só para a solução da odometria visual. Uma abordagem de estimativa de posição probabilística das *features* teve a intenção de otimizar o cálculo dos seis graus de liberdade da câmera.

Com a proposição do PTAM (KLEIN; MURRAY, 2007), foi introduzida a ideia de rastrear a posição da câmera e mapear o ambiente em dois *pipelines* simultâneos. O sistema pode gerenciar milhares de *features* em tempo real, graças ao método de mapeamento separado. Contudo, o método não possuía uma inicialização automática, sendo necessário mover a câmera em direção determinada no início do processamento. Além disso, o número de pontos rastreados ainda se mostrou insuficiente para uso em ambientes de grande escala.

Em (ENGEL; STURM; CREMERS, 2013), os autores propõem uma solução para odometria visual combinada com reconstrução 3D utilizando mapas semi-densos de profundidade inversa. Esses mapas são montados considerando cada profundidade como uma distribuição de probabilidade gaussiana e em cada *frame* a medida é atualizada. O resultado roda em uma CPU em tempo real, porém somente a informação visível da cena é armazenada, eliminando partes não visíveis da cena.

O trabalho de Lim (LIM; LIM; KIM, 2014) propõe um sistema para SLAM monocular de grande escala, utilizando grande quantidade de *features* para mapear o ambiente e tentar diminuir o erro da odometria visual. As *features* são rastreadas utilizando um descritor BRIEF (CALONDER *et al.*, 2010) e a estimativa da posição da câmera é realizada através da minimização do erro de reprojeção. Entretanto, o método não foca na reconstrução do ambiente, e os mapas gerados são somente suficientes para a estimativa da trajetória.

Alguns autores buscaram acelerar partes do algoritmo utilizando hardware específico para isso. (BRENOT; PIAT; FILLATREAU, 2016) constrói um acelerador em hardware para correlacionar *features* BRIEF. Sua intenção é acelerar aplicações de SLAM. Para tanto, a solução é embarcada em uma FPGA em conjunto com uma solução SLAM com-

pleta utilizando dois ARMs em uma placa Zynq. Utilizando 10k LUTs foi possível dobrar o desempenho quando comparado com uma solução sem o acelerador.

A proposta de (LEE; JUNG, 2014) consiste em implementar um acelerador em hardware para resolver mínimos quadrados não lineares usando o método de Levenberg-Marquardt. A solução foca em resolver a matriz de deslocamento da câmera a partir da triangulação de pontos no plano epipolar de duas câmeras. A implementação foi baseada em uma placa KC705 usando 70K LUTs e obtendo um *speedup* de 74,3x na resolução da matriz em comparação com uma implementação em software.

(BOTERO-GALEANO; GONZALEZ; DEVY, 2012) implementa um sistema SLAM baseado em filtro de Kalman estendido em FPGA. O algoritmo de extração de *features* é realizado em tempo real por um hardware rodando na frequência de obtenção dos pixels, enquanto a funcionalidade de construir e atualizar o mapa é feito em software em um PC. Uma implementação hierárquica de software é construída para permitir a transposição do código entre processadores e entre o PC e a plataforma embarcada. A aplicação é capaz de processar 60 *frames* por segundo obtendo resultados satisfatórios de odometria visual.

Outros autores focaram na reconstrução 3D, (NAWAF; TRÉMEAU, 2014) propõe um *framework* que reconstroi o ambiente com ajuda de uma técnica chamada de superpixels. Essa técnica consiste em pré-processar a imagem, segmentando-a em blocos por similaridade. Após, utilizando o fluxo óptico denso, os blocos são considerados planos e posicionados no espaço 3D utilizando uma técnica de *bundle adjustment*. A técnica roda em software em um PC e os resultados mostram a reconstrução de uma cena do dataset KITTI (GEIGER; LENZ; URTASUN, 2012).

Por outro lado, (FIELDS *et al.*, 2009) utiliza informações de um sistema de navegação inercial para obter a posição e orientação da câmera. As informações são agregadas para formar um sistema multi-alcance. O objetivo final é, além da reconstrução de objetos na cena, obter a elevação do terreno com as medidas.

(MEIER *et al.*, 2011) embarcou a solução de localização e reconstrução básica 3D em uma placa que utiliza um ARM7 para fazer o processamento. Sensores de aceleração, velocidade angular e campo magnético são utilizados para complementar os dados de quatro câmeras que são utilizadas para detectar altura, evitar obstáculos e determinar a trajetória do drone. Um filtro de Kalman é utilizado para rastrear a trajetória. Os resultados, apesar de promissores, são bastante preliminares.

Na apresentação do estado da arte, foram mostradas soluções que utilizam implementação puramente em software, ou, em poucos casos, soluções híbridas, conforme pode ser constatado na Tabela 1, que mostra um resumo comparativo das soluções apresentadas no presente Capítulo.

Nessa tabela, as soluções são classificadas por:

- número de câmeras utilizadas; pelo tipo de reconstrução: Esparsa (E), Densa ou Semi-Densa (S) e sem reconstrução (N);

Tabela 1 – Comparativo entre soluções encontradas na literatura e a solução proposta

Sistema	Câmeras	Reconstrução	Tempo Real	Implementação	Plataforma
(NAWAF; TRÉMEAU, 2014)	1	E	N	SW	PC
(ENGEL; STURM; CREMERS, 2013)	1	S	S	SW	PC
(FIELDS *et al.*, 2009)	1	E	N	SW+HWM	PC
(ENGEL; SCHöPS; CREMERS, 2014)	1	S	S	SW	PC
(BRENOT; PIAT; FILLATREAU, 2016)	1	N	S	HWM	FPGA
(DAVISON *et al.*, 2007)	1	N	S	SW	PC
(ENGEL; STüCKLER; CREMERS, 2015)	1	S	S	SW	E+PC
(CIVERA; DAVISON; MONTIEL, 2008)	1	N	S	SW	PC
(LIM; LIM; KIM, 2014)	1	E	S	SW	PC
(LEE; JUNG, 2014)	1	E	S	HWM	FPGA
(BOTERO-GALEANO; GONZALEZ; DEVY, 2012)	1	N	S	HWM	FPGA+PC
(MEIER *et al.*, 2011)	4	N	S	HW+SW	PC E
Sistema Proposto	1	S	S	HW+SW	FPGA

- se a solução permite rodar em tempo real;
- pelo tipo de implementação: Software (SW), Módulo em Hardware (HWM) para ser rodado em outra plataforma e Hardware Dedicado (HW);
- e pelo tipo de plataforma que a solução foi implementada: Computador Pessoal (PC), incluindo soluções que rodam tanto em CPU quanto em GPU, Embarcada (E), Computador Pessoal Embarcado (PC E) e FPGA (FPGA).

Conforme visto, o problema da reconstrução 3D, apesar de possuir na literatura solução em software, ou em algum módulo em hardware que necessita ser acoplado em outra plataforma, ainda carece de uma arquitetura embarcada, que habilite a execução de soluções de visão computacional em tempo real e com baixo consumo.

3.2 MPSoC

Muitos autores propuseram diferentes abordagens utilizando MPSoCs como base para suas soluções. A Empresa francesa Kalray Inc. desenvolveu um produto comercial formado por uma arquitetura *manycore* hierárquica, chamado MPPA-256 (DINECHIN *et al.*, 2013). Nesse sistema são disponibilizados 256 *cores* ao usuários, além de 32 *cores* reservados ao controle do sistema, distribuídos em 16 *clusters*. A comunicação entre *clusters* é realizada através de dois padrões de rede em chip, um para dados e outro para controle.

Cada *cluster* possui até 16 *cores* VLIW e o sistema utiliza uma arquitetura de memória compartilhada. A tecnologia utilizada para produção é de 28nm CMOS, rodando a 400MHz e é focada no mercado de baixo consumo por operação.

O trabalho de (ELMOHR *et al.*, 2018) propõe uma estrutura para gerar automaticamente o código RTL para MPSoCs baseados em rede em chip. Utilizando essa estrutura para gerar seus sistemas, a performance de três configurações de rede em chip em malha são verificadas: 2x2, 4x4 e 8x8. Cada elemento processador do sistema contém um core RI5CY sem periféricos além da interface de rede, que é mapeada diretamente à memória do processador através de um barramento. A arquitetura da rede em chip possui dois barramentos entre os *roteadores*, um para dados outros para controle. O trabalho mostra que o desempenho é melhorado em malhas menores, como esperado, mas a diferença só se torna relevante próximo do ponto onde a rede em chip se torna saturada. Além disso, uma pequena melhora em desempenho durante a saturação é percebida quando os *roteadores* possuem *buffers* maiores. (KHAMIS *et al.*, 2018) utilizou esse trabalho como base, com foco em emulação, construindo uma estrutura completa de verificação. A solução levou a uma simulação mais rápida quando comparada com soluções em software.

Com respeito às estruturas paralelas de processadores, (DAVIDSON *et al.*, 2018) propôs uma estrutura massivamente paralela de processadores RISC-V, dividida em três níveis: um nível contendo um RISC-V RocketChip de cinco *cores* para uso geral, um nível contendo um acelerador utilizando uma rede neural binarizada (BNN) especializada, e uma rede em chip em malha conectando 496 *cores* RISC-V. Cada nível opera em 400MHz, 625MHz e 1,05GHz, respectivamente. Além disso, mais 10 *cores* RISC-V são disponibilizados junto ao conversor DC/DC, utilizando o mesmo código que os outros 496 *cores*. A comunicação entre as entidades é realizada através de um endereçamento global, utilizando o modelo de memória distribuída compartilhada, onde a memória de cada processador é compartilhada globalmente. Isso significa que cada processador pode endereçar a memória de outro processador utilizando o endereço na rede como parte do endereço da memória. A estrutura *manycore* é utilizada para atualizar os pesos da BNN, de forma a acelerar seu acesso à memória, que normalmente é o gargalo dessas redes. Como resultado, mais de 300 FPS foram obtidos processando imagens coloridas de 32x32 pixels.

A plataforma CoreVA-MPSoC é proposta por (SIEVERS *et al.*, 2015), focando em aplicações de *streaming*. A arquitetura é composta por *clusters* ligados em uma rede em chip. Cada *cluster* contém dois ou mais *cores* VLIW conectados entre si por um barramento, utilizando um espaço de endereçamento comum entre eles. Esse trabalho foi expandido por (AX *et al.*, 2018), que propôs três alternativas de organização de memória interna ao *cluster*: memórias locais individuais para cada processador; uma memória compartilhada entre os processadores do mesmo *cluster*; ou uma solução híbrida contendo as duas memórias. Independente da alternativa escolhida, os dados podem ser escritos em

uma memória dentro de um *cluster* por um outro processador pertencente a outro *cluster*, utilizando a capacidade de acesso direto à memória das interfaces de rede. Além disso, um compilador específico para o sistema, levando em conta o paralelismo das aplicações de *streaming*, foi desenvolvido. Nesse trabalho, cada aplicação é considerada estática e o agendamento de tarefas é fixo e definido durante a compilação. A aceleração é avaliada pelo *benchmark* StreamIt, de forma a permitir a comparação com outras soluções. A implementação em silício no conjunto de *standard cell* da ST de 28nm FD-SOI foi realizada e mostra aceleração de até 30x em alguns casos, para uma versão com 32 *cores*.

Uma plataforma MPSoC utilizando rede em chip é proposta por (BAHN; YANG; BAGHERZADEH, 2008) para validar um algoritmo paralelo de FFT de decimação no tempo de radial-2. Cada elemento processador possui um core OpenRISC e memórias de programa e de dados locais. A rede em chip conecta todos os processadores em uma topologia de malha. Cada *roteador* utiliza um controle de fluxo contemplando alguns *wormholes* para reduzir a latência. Na configuração 4x4 da rede em chip, a solução foi capaz de acelerar o cálculo da FFT em até 14,75x para um vetor de 16384 pontos. Entretanto, o desempenho é bastante inferior em vetores menores, como a aceleração de apenas 4,84x em um vetor de 128 pontos.

Uma arquitetura para obter o perfil de desempenho de diferentes implementações de FFT utilizando o MPSoC Ninesilica é utilizado por (AIROLDI; GARZIA; NURMI, 2010). O MPSoC Ninesilica é um modelo que permite a criação de diferentes arquiteturas. Cada elemento processador, que é conectado diretamente à rede em chip em malha, possui um processador e uma memória local para realizar os cálculos. Nove desses elementos processadores são dispostos numa rede em chip 3x3, onde o elemento central contém a interface externa do sistema e é utilizado como gerente do sistema. O sistema é capaz de acelerar em até 5,9x o processamento da FFT de 64 pontos, e em até 6,9x o processamento da FFT de 2048 pontos.

Um método de múltiplas *threads* desenvolvido para rodar em MPSoCs baseados em NoC de memória privada distribuída é proposto por (GARIBOTTI *et al.*, 2013). Cada elemento processador executa seu próprio sistema operacional de tempo real independentemente. A sincronização entre eles é feita utilizando um protocolo de troca de mensagens através da rede em chip. Um protocolo em hardware dedicado foi desenvolvido para interligar as memórias privadas entre si. Um mecanismo de controle de cache que opera sobre a rede em chip mantém cópias de memória local de outros processadores. As aplicações testadas foram Smith Waterman (utilizado para correspondência de DNA), codificação MJPEG e FFT. Utilizando uma configuração de rede em chip 3x3, com cada processador contendo 16KB de memória cache, a solução obteve aceleração de 4,6x para a FFT.

A Tabela 2 mostra as arquiteturas propostas considerando os seguintes aspectos:

- Frequência de Operação em MHz;

- Número total de cores do sistema;
- Tipo de core utilizado;
- Organização da NoC;
- Se os *cores* são organizados em *clusters*, quantos *cores* são agrupados em cada *cluster*;
- Forma de implementação do MPSoC

Tabela 2 – Comparativo entre MPSoCs encontrados na literatura e o proposto

Sistema	Freq. (MHz)	# de *cores*	Tipo de *core*	Tipo NoC	*Cluster*	Plataforma
(GARIBOTTI *et al.*, 2013)	500	9	Microblaze	3x3	-	FPGA
(AIROLDI; GARZIA; NURMI, 2010)	180	9	COFFEE RISC	3x3	-	FPGA
(BAHN *et al.*, 2008)	100	4 16 64	OpenRISC	2x2 4x4 8x8	-	Simulação
(SIEVERS *et al.*, 2015)	830	16	VLIW	2x2 2x1	4 8	ASIC
(AX *et al.*, 2018)	650	32	VLIW	4x2 2x2 2x1	4 8 16	ASIC
(DINECHIN *et al.*, 2013)	400	256	VLIW	4x4	14	ASIC
(ELMOHR *et al.*, 2018)	250	4 16 64	RISC-V	2x2 4x4 8x8	-	Simulação
(DAVIDSON *et al.*, 2018)	1050	496	RISC-V	31x16	-	ASIC
MPVue	50	9 16	RISC-V	3x3 4x4	-	FPGA

Na apresentação de revisão de literatura dos MPSoCs, foram mostradas soluções que utilizam diferentes organizações internas, diferentes tipos e quantidades de *cores* e rodando a diferentes frequências. Entretanto, nota-se que ainda existe a necessidade de desenvolvimento de uma solução que consiga utilizar todo o potencial do sistema como um todo.

Dentre as soluções mostradas na tabela, foram selecionadas as soluções de (GARIBOTTI *et al.*, 2013; AIROLDI; GARZIA; NURMI, 2010; BAHN *et al.*, 2008; SIEVERS *et al.*, 2015; AX *et al.*, 2018) para serem comparadas com o sistema proposto, chamado MPVue.

3.3 Conclusão do Capítulo

Os estudos do estado da arte foram realizados e diversas soluções resolvem o problema de reconstrução 3D. Todavia, ainda é necessário o desenvolvimento de uma solução que possa ser embarcada diretamente no hardware onde a câmera se encontra. Os sistemas propostos por outros autores foram então apresentados e suas características foram levantados para embasar a comparação com o sistema proposto.

4 MATERIAIS E MÉTODOS

Nesse capítulo, são descritos os ambientes de desenvolvimento das diversas etapas do trabalho, desde a prototipação em alto nível, utilizando a linguagem Python, até o desenvolvimento da solução final, sintetizada em FPGA e rodando software embarcado.

4.1 Desenvolvimento de Protótipo

Para validar o algoritmo proposto, as etapas foram implementadas em script Python e executadas em um computador Intel i7 de segunda geração com 8GB de RAM, utilizando a plataforma Anaconda Spyke[1]. A biblioteca OpenCV[2] (BRADSKI, 2000) foi utilizada para aqueles algoritmos que já estavam implementados e disponibilizados. Sob tal panorama, o processo de implementação e debug pode ser acelerado. O teste se deu em situações e vídeos reais, assim como em datasets criados para esse fim, como, por exemplo, o KITTI[3] (GEIGER; LENZ; URTASUN, 2012) – que possui 22 sequências de imagens extraídas a partir de um veículo em movimento preparado para esse fim, sendo que 11 dessas sequências estão disponíveis com as informações de movimento da câmera – ou o LSD-SLAM[4] (ENGEL; SCHöPS; CREMERS, 2014). Ademais, a solução é analisada com base nos resultados obtidos na submissão ao 3D-RMS[5] (TYLECEK *et al.*, 2019), desafio de reconstrução 3D que ocorreu junto ao ECCV 2018 Workshop.

O software foi analisado pela ferramenta de profiling da própria suíte Anaconda, que fornece detalhamento a nível de função, para analisar o desempenho da solução de forma aprofundada.

4.2 Desenvolvimento de Hardware e Software Embarcado

Alguns trabalhos de outros autores foram utilizados como base para o desenvolvimento do Hardware. O core do processador foi baseado no *core* RI5CY, da plataforma

[1]Disponível em:https://anaconda.org/anaconda/python
[2]Disponível em: https://opencv.org/
[3]Disponível em: http://www.cvlibs.net/datasets/kitti/
[4]Disponível em: https://github.com/tum-vision/lsd_slam
[5]Disponível em: http://trimbot2020.webhosting.rug.nl/events/3drms/challenge/

PULP, que foi introduzida por (CONTI *et al.*, 2014), visando o processamento paralelo de baixo consumo. Nesse modelo. diversos *cores* do tipo OpenRISC são utilizados para realizar a computação de redes neurais convolutivas (CNN). Um controle para alternar entre baixa e alta frequência é utilizado para mudar o comportamento do sistema quando eventos específicos ocorrem. Essa plataforma foi atualizada por (TRABER *et al.*, 2016) para utilizar RISC-V e se apresenta como uma alternativa de baixo consumo para Internet das Coisas. Ela reúne algumas arquiteturas de processador em conjunto com periféricos para permitir projeto de SoC de forma rápida e confiável. Quatro *cores* RISC-V de código aberto estão disponíveis: RI5CY, Micro riscy, zero riscy e Ariane (ETH Zurich, 2019). O RI5CY é um *core* RISC-V de 32 bits com pipeline de quatro estágios, que inclui as extensões de instruções para cálculos com números inteiros, multiplicação e instruções compactas, chamado de RV32IMC (TRABER; GAUTSCHI; SCHIAVONE, 2016). Além disso, o conjunto de instruções foi ampliado para suportar instruções adicionais direcionadas para aplicações de processamento de sinais, como controle em hardware para laços, carga e salvamento com pós-incremento, e instruções adicionais de ULA. Os *cores* são detalhados na descrição da arquitetura, na Seção 7.3.

Já o desenvolvimento da rede em chip foi baseado num trabalho desenvolvido anteriormente no laboratório. Com o intuito de melhorar a segurança de MPSoCs, (REINBRECHT *et al.*, 2016) propôs uma rede em chip que possui resistência a ataques de análise diferencial de consumo (DPA), chamada Gossip NoC. Tal rede em chip possui topologia em malha com roteamento XY, que é alternado para YX no caso de que um ataque ser detectado. Ademais, a rede possui suporte para memória local ou compartilhada. Uma extensão para prevenir ataques de canal lateral é fornecida através de uma conexão adicional entre os *roteadores*, chamada *gossip*. Cada vez que um *roteador* identifica uma possível ameaça no sistema, a informação é enviada a seus vizinhos, quando o nível de *gossip* chega a um limiar, o roteamento é alterado. Essa solução foi capaz de reduzir falsos positivos na detecção de ameaças na rede. Essa arquitetura foi utilizada como base para o desenvolvimento do presente trabalho. Seu detalhamento e características são mostradas na Seção 7.2.

A arquitetura em hardware foi implementada em RTL sintetizável, em linguagem Verilog e System Verilog, e roda em FPGA. A placa escolhida foi a KC705 (Kintex-7 XC7K325T) da Xilinx, que possui 478K células lógicas e 445 blocos de RAM de 36Kb cada, totalizando 16Kb para uso do sistema. A síntese foi realizada no ambiente Vivado 2018.3, utilizando scripts Verilog RTL puros e os IPs necessários, como os PLLs e *phys* para UART e Ethernet.

O software embarcado pode ser compilado por qualquer compilador compatível com o conjunto de instruções do RISC-V, opcionalmente configurado para utilizar as instruções contidas nas extensões: multiplicação e divisão (M) e instruções compactas (C). Se as instruções específicas para o processador forem usadas, é necessário utilizar um com-

pilador GCC específico para ele, tornando o uso das funções aceleradas transparente ao programador. O desenvolvimento foi feito utilizando a IDE Eclipse integrado com o conjunto de ferramentas GNU MCU Eclipse[6] (IONESCU, 2017), que contém compilador, linker e montador para RISC-V. Para realizar a simulação do software em alto nível o simulador Spike[7] (SPIKE, A RISC-V ISA Simulator., 2019) foi utilizado. A integração entre o simulador e o Eclipse é fornecida através das interfaces GNU Debugger (GDB) e um controlador de JTAG, como o Open On-Chip Debugger (OpenOCD).

4.3 Análise de Desempenho Paralelo

Em sistemas de processamento paralelo, diversos fatores podem influenciar no desempenho do sistema, como frequencia de operação, número de processadores disponíveis, entre outros. Para que os sistemas sejam comparáveis entre si, algumas técnicas foram propostas para normalizar os resultados e permitir conclusões sobre os diferentes aspectos de projeto de um sistema embarcado de processamento paralelo.

4.3.1 Aceleração

Para que a comparação entre sistemas de processamento paralelo seja independente da frequência de operação dos sistemas, o conceito de aceleração é utilizado. O método consiste em dividir o tempo de execução do algoritmo rodando em apenas um dos processadores do sistema, pelo tempo de execução rodando no sistema completo. Dessa forma, mede-se o quanto mais rápido o sistema é quando comparado com um sistema não-paralelo. Além disso, permite a comparação de sistemas, desde que ambos possuam a mesma quantidade de processadores.

4.3.2 Eficiência

Quando dois sistemas com quantidades de processadores diferentes precisam ser comparados, pode-se utilizar o método da eficiência, proposto por Airoldi (AIROLDI; GARZIA; NURMI, 2010). Ela é obtida dividindo-se a aceleração pelo número de elementos processadores.

Dessa forma, mede-se quão eficientemente o sistema utiliza seus processadores durante a execução do algoritmo.

4.3.3 Bloodgraph

Como proposto pela Lei de Amdahl (SHI, 2013), um dos aspectos que degrada o desempenho de um sistema multiprocessado é a parte do algoritmo que não pode ser paralelizado. Isso significa que esse código deve ser executado em um único processador,

[6]Disponível em: https://gnu-mcu-eclipse.github.io/
[7]Disponível em: https://github.com/riscv/riscv-isa-sim

enquanto os outros processadores aguardam seu término. Apesar dos algoritmos selecionados para comparação não possuírem nenhuma parte explícita não-paralelizável, alguma execução serial é realizada durante a preparação dos dados para processamento e o seu retorno dos processadores após a tarefa ser executada.

De forma a medir a perda de desempenho por conta de execução de código serial e outras ineficiências, a técnica de *bloodgraph* baseada em nos trabalhos de (DUVALL, 2004; JOHNSSON; STENEMO; ABDIN, 2005) foi utilizada. Essa técnica mostra os eventos dos processadores em uma linha do tempo, mostrando seu estado durante o processamento. Para permitir a extração de dados relevantes, sem consumir excessivamente os recursos dos processadores, quatro estados foram escolhidos: *IDLE* – aguardando uma nova tarefa ou aguardando outro processador terminar sua tarefa; *COMM* – comunicando com outro processador; *TASK* – executando uma tarefa; *CONTROL* – quando o LaPSIman está preparando uma tarefa para ser enviada a outro processador. Toda vez que um processador troca seu estado, o *timestamp* é armazenado em conjunto com o novo estado. No fim da computação, os dados são extraídos de cada processador e o *bloodgraph* é consolidado.

5 SOLUÇÃO PROPOSTA PARA RECONSTRUÇÃO 3D

Portanto, é proposta uma plataforma de projeto para o desenvolvimento de sistemas integrados software-hardware (SW/HW codesign) específicos. Essa plataforma habilita a criação de solução de visão computacional, como, por exemplo, algoritmo capaz de identificar o ambiente à sua volta, viabilizando que, a partir da localização do observador, possa se construir a nuvem de pontos semi-densa de onde seja possível identificar a presença de objetos e suas distâncias.

Adicionalmente, vislumbrando a implementação desse algoritmo de reconstrução 3D para ser executado em um sistema embarcado, e para contribuir com o estado da arte, é proposta uma nova solução para reconstrução 3D, a partir das imagens e dos dados de movimento de uma única câmera, conforme descrito nesse capítulo.

Para tanto, uma nova forma de cálculo de profundidade é proposta, contendo operações mais simples. Além disso, desse cálculo deriva-se uma estimativa de precisão da medida.

O perfil de desempenho desse software foi estudado para identificar as funções com potencial de otimização ou que podem ser paralelizadas, como, por exemplo, funções simples que rodam múltiplas vezes.

A arquitetura de hardware proposta, consiste em uma plataforma que pode ser implementada em FPGA ou em ASIC, conforme descrito no Capítulo 7, contendo múltiplas unidades de processamento. Por fim, tal metodologia é focada no reuso, implementando-se a solução de hardware em IPs e a solução de software em bibliotecas, oferecendo uma arquitetura totalmente integrada.

5.1 Premissa

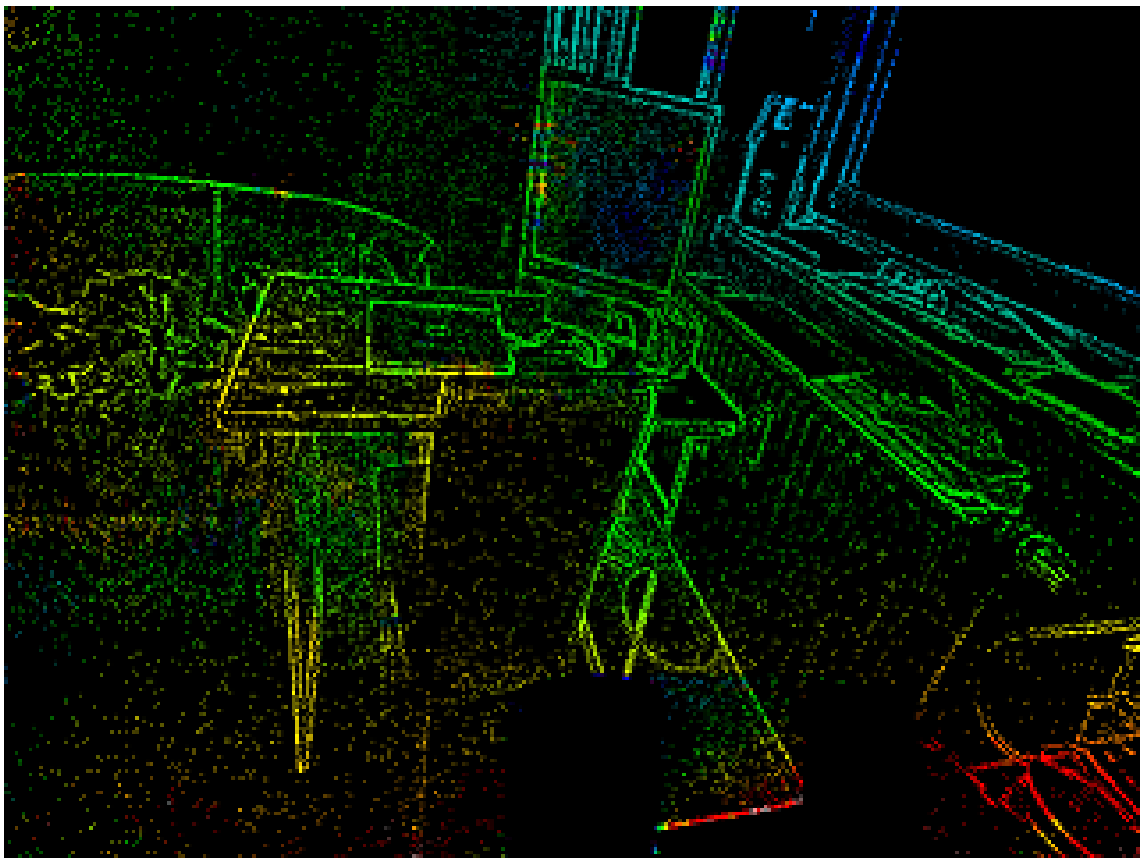
Com a evolução dos algoritmos esparsos de odometria visual, é possível estimar, com grande precisão, o movimento da câmera, mesmo com poucas *features* sendo rastreadas (PEREIRA *et al.*, 2017, 2018). Não obstante, os algoritmos utilizados são computacionalmente complexos para realizar a reconstrução densa ou semi-densa do ambiente ao mesmo tempo em que estima a localização da câmera.

A solução proposta parte do princípio de que a posição da câmera é conhecida – ou que já foi calculada em um passo anterior – e tenta mapear o ambiente ao redor do observador. O sistema deve, ao mesmo tempo, tomar cuidado com as restrições típicas de um sistema embarcado, como consumo de memória, frequência máxima de operação e limitação de operações complexas, como números em ponto flutuante, ou trigonométricas.

5.2 Primeiros Testes

Nos primeiros testes realizados, a detecção do movimento da câmera ($\vec{\Theta} = (\vec{T}, \vec{\Omega})$) foi realizada ao mesmo tempo que o cálculo de profundidade. Os pontos eram escolhidos como *features* pelos seus valores de gradiente (CANNY, 1987). A Figura 2 mostra o mapa de profundidades, calculado sobre uma sequência de *frames* do conjunto LSD-SLAM (ENGEL; SCHöPS; CREMERS, 2014). Os valores de profundidade ($\rho(x, y)$) são representadas em uma escala de vermelho (mais perto) a roxo (mais distante).

Figura 2 – Mapa de profundidades. Sequência: LSD-SLAM, *frames* 1 ao 5



Fonte: do autor

Nesse teste, os valores de profundidade foram iniciados com valores aleatórios, então se calculou alternadamente o movimento da câmera, seguido do cálculo da profundidade, com auxílio da Equação (5). O processo foi repetido até que o resultado convergisse. Apesar do resultado visual ser interessante, a presença de *outliers* fez com que o algoritmo

não obtivesse convergência quando aplicado a uma sequência maior de *frames*. Ademais, o tempo de execução dessa solução tornou-o impraticável. Eram necessários alguns minutos para se calcular os valores de profundidades de um único *frame*.

Com isso, soluções mais robustas tiveram de ser buscadas. A implementação separou, a partir desse momento, o cálculo do movimento da câmera do cálculo de profundidade.

5.3 Cálculo de Profundidades

Para se estimar a profundidade dos pontos de uma imagem – ou de um conjunto de imagens – um grande esforço computacional precisa ser empregado. As soluções propostas por outros autores não são apropriadas para serem embarcadas. Conforme mostrado por (STRASDAT; MONTIEL; DAVISON, 2010), as soluções de EKF-SLAM – utilizadas por diversos autores, como (CIVERA; DAVISON; MONTIEL, 2008; HUANG; MOURIKIS; ROUMELIOTIS, 2008) – possuem custo computacional de $O(n^2)$, onde n é o número de pontos a serem rastreados por *frame*. Por outro lado, as soluções de Bundle Adjustment (BA) – como usado por (LIM; LIM; KIM, 2014) – incluem a solução de mínimos quadrados não linear, com a solução de Jacobianos, o que também possui alto custo computacional.

Nessa seção, é proposta uma nova forma de estimar a profundidade de um ponto em uma cena, considerando que ele tenha sido observado por N *frames*, consecutivos ou não, desde que seja possível determinar a posição desse ponto em todos os *frames* e o movimento relativo da câmera entre os *frames*. O valor de profundidade pode ser obtido utilizando basicamente multiplicações, divisões, somas e subtrações. Apenas na última etapa é necessário o uso de raiz quadrada.

5.3.1 Descrição da Metodologia de Cálculo

Segundo (HEEGER; JEPSON, 1992) a diferença de posição $\vec{\Delta}(x, y)$ de um ponto projetado em uma imagem, em comparação a uma imagem de referência, depende do movimento de translação $\vec{T}$ e rotação $\vec{\Omega}$ da câmera, e pode ser calculado pela Equação (5). Contudo, o valor de $\vec{\Delta}(x, y)$ normalmente é medido através do rastreamento de *features* entre imagens consecutivas. A informação faltante na imagem é, então, a profundidade das *features*. Por consequência, a incógnita da Equação (5) acaba por ser ρ.

Além de um método para o cálculo de profundidade de pontos, novas métricas de estimativa de erro são descritas para agregar a métrica de definição de pontos bons de *outliers*. Para tanto, os dados são organizados na forma apresentada pelas equações (15), (16) e (17). As observações O_i^N são as coordenadas da *feature* rastreada a partir do *frame* i por N *frames* consecutivos. $\vec{\Omega}_i^{N-1}$ e $\vec{T}_i^{N-1}$ são, respectivamente, a rotação e a translação da câmera entre o *frame* i e os $N-1$ *frames* consecutivos.

$$O_i^N = \begin{bmatrix} x_i & x_{i+1} & x_{i+2} & ... & x_{i+N-1} \\ y_i & y_{i+1} & y_{i+2} & & y_{i+N-1} \end{bmatrix} \tag{15}$$

$$\vec{\Omega}_i^{N-1} = \begin{bmatrix} \Omega_{x_{i,i+1}} & \Omega_{x_{i,i+2}} & & \Omega_{x_{i,i+N-1}} \\ \Omega_{y_{i,i+1}} & \Omega_{y_{i,i+2}} & ... & \Omega_{y_{i,i+N-1}} \\ \Omega_{z_{i,i+1}} & \Omega_{z_{i,i+2}} & & \Omega_{z_{i,i+N-1}} \end{bmatrix} \tag{16}$$

$$\vec{T}_i^{N-1} = \begin{bmatrix} T_{x_{i,i+1}} & T_{x_{i,i+2}} & & T_{x_{i,i+N-1}} \\ T_{y_{i,i+1}} & T_{y_{i,i+2}} & ... & T_{y_{i,i+N-1}} \\ T_{z_{i,i+1}} & T_{z_{i,i+2}} & & T_{z_{i,i+N-1}} \end{bmatrix} \tag{17}$$

Nota-se que, como a medida de translação e rotação é realizada entre *frames*, há uma medida a menos do que as observações. A matriz de observações possui tamanho $2 \times N$, enquanto as matrizes de movimento possuem tamanho $3 \times N - 1$ cada. Assim, a primeira observação da *feature* no *frame* i ($O_i^1 = [x_i, y_i]^T$) é considerada a posição de referência.

Conforme mostra a Equação (18), $\bar{\vec{\Delta}}_{i+1}^{N-1}$ pode ser medido pela subtração da posição da *feature* rastreada nas demais imagens pela sua posição na imagem de referência. Dessa forma, $\bar{\vec{\Delta}}_{i+1}^{N-1}$ possui tamanho $2 \times N - 1$, e possui a mesma quantidade de amostras que $\vec{\Omega}_i^{N-1}$ e $\vec{T}_i^{N-1}$

$$\bar{\vec{\Delta}}_{i+1}^{N-1} = O_{i+1}^{N-1} - O_i^1 \tag{18}$$

Ainda, as equações (6) e (7), que definem as matrizes A e B da Equação (5), dependem apenas da posição de referência da *feature* na origem do movimento. Desse modo, A e B devem ser calculadas apenas uma vez por *feature*.

Conhecendo os valores do fluxo óptico e do movimento da câmera, definidos nas equações (16), (17) e (18), a Equação (5) pode ser reescrita de forma matricial para a *feature* como:

$$\bar{\vec{\Delta}}_{i+1}^{N-1} - B \cdot \vec{\Omega}_i^{N-1} = \rho(A \cdot \vec{T}_i^{N-1}) \tag{19}$$

As matrizes que possuem valores constantes podem ser reduzidas em duas matrizes $\vec{\alpha}_{i+1}^{N-1}$ e $\vec{\beta}_{i+1}^{N-1}$, definidas como:

$$\vec{\alpha}_{i+1}^{N-1} = A \cdot \vec{T}_i^{N-1} \tag{20}$$

$$\vec{\beta}_{i+1}^{N-1} = \bar{\vec{\Delta}}_{i+1}^{N-1} - B \cdot \vec{\Omega}_i^{N-1} \tag{21}$$

Aplicando as equações (20) e (21) na Equação (19), obtém-se a equação de medida de paralaxe:

$$\vec{\beta}_{i+1}^{N-1} = \rho(\vec{\alpha}_{i+1}^{N-1}) \tag{22}$$

Após as multiplicações matriciais, $\vec{\alpha}_{i+1}^{N-1}$ e $\vec{\beta}_{i+1}^{N-1}$ possuem tamanho $2 \times N - 1$ e ρ é um escalar. $\vec{\beta}_{i+1}^{N-1}$ representa o movimento medido da *feature* descontando o efeito da rotação da câmera. Esse efeito é chamado de paralaxe, e o seu valor pode ser estimado por $\rho(\vec{\alpha}_{i+1}^{N-1})$ considerando a translação da câmera e a profundidade do ponto. Como o movimento de paralaxe ocorre no plano da imagem, ele possui uma componente em x e uma em y, é possível, então, calcular o valor absoluto de cada termo:

$$|\vec{\alpha}| = \sqrt{(\alpha_x)^2 + (\alpha_y)^2} \qquad |\vec{\beta}| = \sqrt{(\beta_x)^2 + (\beta_y)^2} \tag{23}$$

Aplicando-se a Equação (23) na Equação (22) obtém-se:

$$|\vec{\beta}_{i+1}^{N-1}| = \rho|\vec{\alpha}_{i+1}^{N-1}| \tag{24}$$

A Equação (24) é formada por duas matrizes $|\vec{\beta}_{i+1}^{N-1}|$ e $|\vec{\alpha}_{i+1}^{N-1}|$ de tamanho $1 \times N - 1$ e o escalar ρ.

Finalmente, ρ pode ser resolvido por uma regressão linear simples, com apenas um regressor.

$$\rho = \frac{\sum_{j=i+1}^{N-1} |\vec{\alpha}_j^1||\vec{\beta}_j^1|}{\sum_{j=i+1}^{N-1} (|\vec{\alpha}_j^1|)^2} \tag{25}$$

Além disso, num ambiente embarcado, as raízes quadradas da Equação (23) não necessitam ser imediatamente calculadas, podendo ser levados os valores intermediários para a Equação (25) e realizar a extração de apenas uma raiz quadrada por termo da soma. Gera-se, assim, economia de operações para o processador.

Nesse sentido, o valor da profundidade do ponto pode ser calculado utilizando-se apenas os valores rastreados da *feature* que o representa e os vetores de movimento da câmera. Apenas o valor absoluto do movimento de paralaxe é utilizado, desprezando-se, num primeiro momento, a informação angular. Ela será utilizada em seguida, como uma medida da qualidade da estimativa.

5.3.2 Estimativa de erro

Conforme visto antes, os termos presentes na Equação (19), que mede o deslocamento de paralaxe, tem componente em x e em y. Na Equação (23), o valor absoluto é calculado, desprezando a informação angular do movimento de paralaxe, que não é necessária para o cálculo da profundidade. Todavia, essa informação angular pode trazer informação interessante a respeito do erro das medidas, uma vez que ângulos discrepantes entre a medida e o esperado podem indicar que a estimativa como um todo está comprometida. Logo, o cálculo do ângulo do deslocamento de paralaxe medido e estimado pode ser

obtido:

$$\angle\vec{\alpha} = \text{atan2}(\alpha_x, \alpha_y) \qquad \angle\vec{\beta} = \text{atan2}(\beta_x, \beta_y) \tag{26}$$

Como ρ é um escalar, ele não tem valor de ângulo. Então, a semelhança entre os dois ângulos é diretamente proporcional à qualidade dos valores de entrada da função de cálculo de profundidade. Dessa forma, a diferença entre ambos é obtida

$$\gamma = \angle\vec{\alpha} - \angle\vec{\beta} \tag{27}$$

Em sistemas embarcados, onde não se dispõe de grande poder computacional para calcular funções trigonométricas, pode-se utilizar a identidade trigonométrica de subtração:

$$\begin{aligned} \gamma &= \angle\vec{\alpha} - \angle\vec{\beta} \\ &= \text{atan2}(\alpha_x, \alpha_y) - \text{atan2}(\beta_x, \beta_y) \\ &= \text{atan2}(\alpha_x\beta_y - \beta_x\alpha_y, \alpha_y\beta_y + \alpha_x\beta_x) \end{aligned} \tag{28}$$

Ainda, se a diferença entre os ângulos for pequena, pode-se aproximar a Equação (28) pela termo de primeira ordem de sua série de Taylor:

$$\gamma \approx \frac{\alpha_x\beta_y - \beta_x\alpha_y}{\alpha_y\beta_y + \alpha_x\beta_x} \tag{29}$$

A medida de γ é uma matriz de tamanho (1xN-1), que representa o erro em ângulo entre o valor medido e o valor estimado de deslocamento de paralaxe. Essa matriz é muito grande para ser armazenada ou manipulada por outras etapas do algoritmo. Dessa forma, os valores de γ foram considerados com distribuição gaussiana, e apenas sua média μ_γ e o desvio padrão σ_γ são utilizados em conjunto com o valor de profundidade ρ.

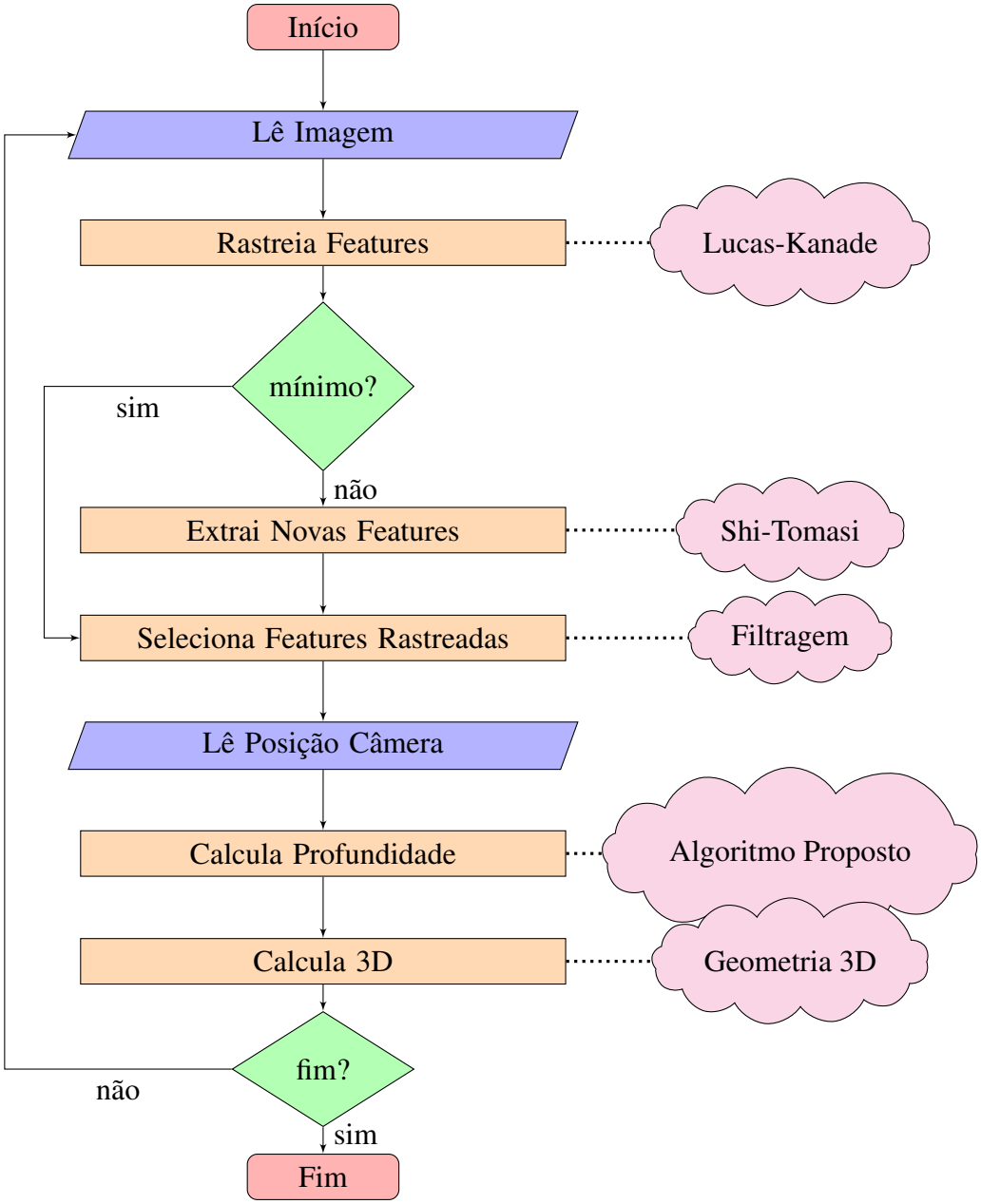
5.4 Algoritmo Proposto para Reconstrução 3D

Nessa seção, o procedimento de reconstrução 3D baseado na posição da câmera é proposto. Ele será utilizado como base para as implementações em software no restante do presente trabalho. Dessa maneira, os conceitos apresentados são utilizados de forma a tornar o sistema mais eficiente.

O fluxograma básico do algoritmo pode ser visto na Figura 3. Nele, o início e o fim são representados pelos blocos vermelhos, os procedimentos pelos laranjas, as entradas de dados pelos azuis e as decisões pelos verdes. As etapas são melhores explicadas a seguir.

Na etapa *Lê Imagem*, a imagem é recebida pelo sistema e armazenada na memória.

Figura 3 – Fluxograma da Recontrução 3D



Fonte: do autor

Essa recepção pode ser de um fluxo de dados de uma câmera, por exemplo, ou pode ser simplesmente uma leitura de uma memória externa. Duas imagens devem ser armazenadas, para que possam ser comparadas. Na etapa *Rastreia Features*, as *features* são rastreadas entre o *frame* atual e o anterior, utilizando o algoritmo de Lucas-Kanade (BOUGUET, 2000). Aquelas que forem encontradas são armazenadas, e as restantes, descartadas. O processo de rastreamento é detalhado na Seção 6.1. A seguir, se não houver uma quantidade mínima de *features* rastreadas, um novo conjunto de *features* é gerado na etapa *Extrai Novas Features*, utilizando o algoritmo de Shi-Tomasi chamado Good Features to Track (SHI; TOMASI, 1994) tendo sua implementação detalhada na Seção 6.2. As *features* que foram rastreadas por um mínimo de *frames* seguidos são selecionadas em *Seleciona Features Rastreadas*, utilizando um algoritmo simples de filtragem. Após, a profundidade e a posição 3D das *features* são calculadas nas etapas *Calcula Profundidade* e *Calcula 3D*, respectivamente. Para se calcular a profundidade, o algoritmo proposto na Seção 5.3 é utilizado. Sua implementação é contextualizada na Seção 6.3. Já o cálculo da posição 3D utiliza os conceitos utilizados na geometria 3D e são melhor explicados na Seção 6.4. Por fim, se houverem mais imagens a serem processadas, o processo se repete, ao contrário, o processo se encerra.

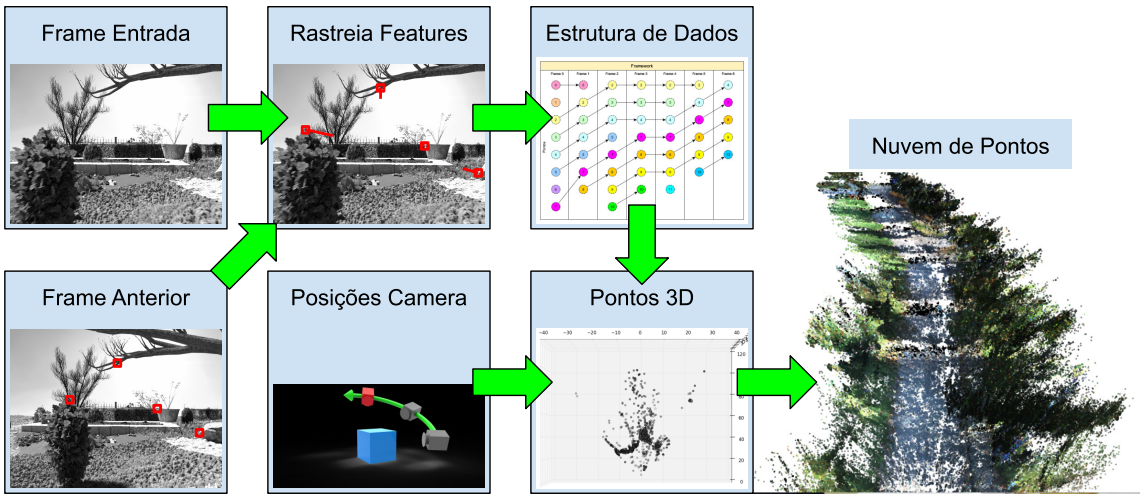
5.5 Conclusão do Capítulo

A solução para reconstrução 3D foi proposta, comparável com as soluções do estado da arte. As premissas foram apresentadas, bem como testes foram realizados para contextualizar o problema. Em seguida, uma metodologia para calcular a profundidade de pontos em uma sequencia de imagens é proposta. Por fim, o fluxograma da solução foi exibido.

6 IMPLEMENTAÇÃO DE PROTÓTIPO

Nesse capítulo são mostrados os detalhes da implementação e os resultados parciais obtidos, de forma a demonstrar a evolução do trabalho. Seguindo a sequência determinada pelo fluxograma apresentado no capítulo anterior, podemos determinar o fluxo normal dos dados da solução como o apresentado na Figura 4. Seguindo esse fluxo, são mostrados no capítulo o processo de Rastreamento de Features, de Extração de Novas Features, o Cálculo de Profundidades e a transformação de pontos no plano da imagem em pontos no espaço 3D, incluindo os algoritmos utilizados e suas restrições. Além disso, é descrita a estrutura de dados utilizada, seguido do método de exportação dos dados. Após, são apresentados os resultados e uma análise de erros. Por fim, uma análise do desempenho é realizada para nortear as escolhas realizadas durante a implementação do sistema embarcado.

Figura 4 – Fluxo de Dados da Solução



Fonte: do autor

6.1 Rastreamento de Features

De forma a iniciar a implementação do algoritmo proposto, o rastreamento de *features* foi desenvolvido. Em cada *frame*, *features* do *frame* anterior são rastreadas utilizando o algoritmo de Lucas-Kanade, descrito nas equações (13) e (14), através do método proposto por (BOUGUET, 2000), conforme descrito na Subseção 2.2.2. É utilizada a técnica de procura piramidal das regiões de interesse. A imagem é recursivamente sub-amostrada, facilitando o cálculo do deslocamento das regiões de interesse, pois o deslocamento é pequeno nas imagens sub-amostradas. Durante o retorno do cálculo, o refinamento do valor é feito em cada nível, alcançando, assim, a precisão desejada.

Esse método está disponível na OpenCV e é possível selecionar os parâmetros de tamanho de janela e altura da pirâmide. Os valores dos parâmetros foram intrinsicamente ajustados para melhor desempenho nas imagens do dataset do KITTI. Desse modo, foi escolhida o tamanho da janela de 13x13 para comparação das *features*, a altura da pirâmide com quatro níveis e o máximo de iterações em 50 vezes.

6.2 Extração de Novas Features

Caso nem todas as *features* tenham sido encontrados na etapa de rastreamento, novas *features* são adicionadas, utilizando o método Good Features to Track (SHI; TOMASI, 1994), mantendo, assim, a quantidade de *features* a serem procurados por *frame* constante.

Para a inicialização do algoritmo, considera-se que não há *features* para serem rastreadas do *frame* anterior. Dessa forma, todas as *features* no primeiro *frame* são obtidas a partir do algoritmo para Extração de Novas *features*. A quantidade de *features* a serem utilizadas pode ser definida no início do processamento e pode variar de algumas unidades às centenas de milhares. A escolha desse número depende da performance que se deseja, pois o tempo de processamento do *frame* depende inerentemente do número de *features* rastreadas *versus* a qualidade da reconstrução desejada, já que, quanto mais *features* forem rastreados, mais rica em detalhes é a reconstrução do ambiente.

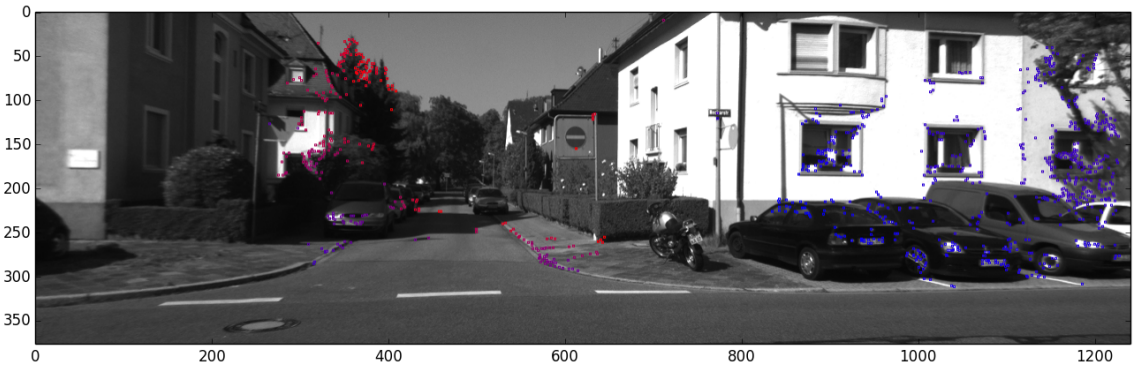
Para evitar a formação de grupo de pixels muito próximos durante a extração se limitou a distância entre *features* de no mínimo 1 pixel.

6.3 Cálculo de Profundidade

A profundidade de cada *feature* é, então, calculada conforme o método proposto na Seção 5.3. Após a utilização desse método, a Figura 5 mostra um *frame* da sequência KITTI (GEIGER; LENZ; URTASUN, 2012) com os valores de profundidade ($\rho(x, y)$) dos pontos p_c pintadas em uma escala de azul (mais perto) a vermelho (mais distante). É possível notar a consistência visual do algoritmo, pois os pontos detectados na parede do prédio branco à direita possuem uma faixa de *cores* homogênea, indicando distâncias do

observador muito parecidas, o que condiz com o fato de pertencerem à mesma parede. É importante notar também a mudança de cor dos pontos detectados no cordão da calçada, mais ao centro da imagem. Essa variação indica que, conforme os pontos vão ficando mais vermelhos, eles estão mais longe do observador. Esse resultado é consistente com o a cena e pode ser constatado visualmente.

Figura 5 – Imagem com profundidades. KITTI - Sequência 0, *frames*: 20 ao 30



Fonte: do autor

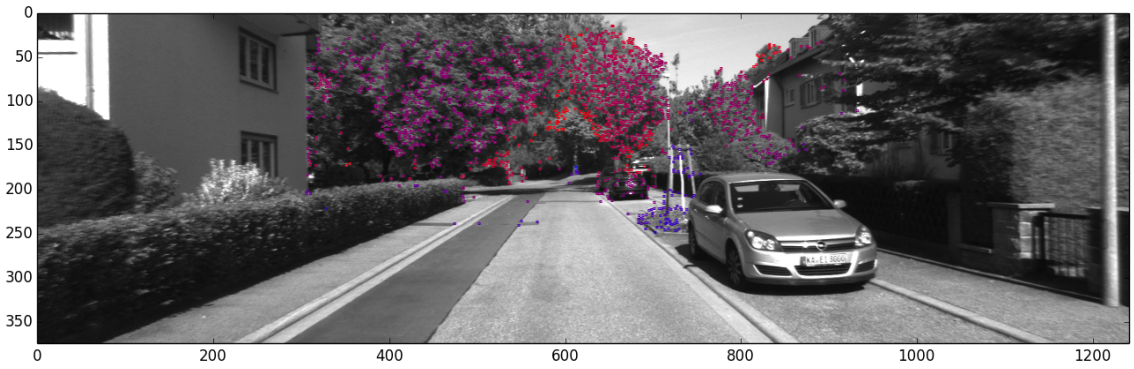
A partir da informação de profundidade, é possível calcular as coordenadas de cada um dos pontos para o sistema de coordenadas da câmera, utilizando a Equação (8). A Figura 6 mostra um *frame* da sequência KITTI com as profundidades plotadas e a Figura 7 mostra os mesmos pontos convertidos ao sistema de coordenadas da câmera (P_c), em uma plotagem 3D. Nela, os pontos são vistos de cima, com o eixo vertical mostrando a profundidade. Nessa figura é possível notar o aglomerado de pontos que formam as árvores mostradas na figura anterior. Os pontos que compõem a estrutura atrás do carro também estão em destaque, em frente às árvores. Nota-se, também, os pontos pertencentes às árvores, à esquerda, visualmente coerentes com a imagem mostrada. Mesmo assim, é possível notar a presença de alguns *outliers*, que representam um desafio a ser superado pelo algoritmo quando da montagem do ambiente 3D.

6.4 Pontos em 3D

A partir da posição de uma *feature* (x, y) e sua profundidade ρ é possível utilizar a Equação (8) para se calcular a posição de um ponto em relação ao sistema de coordenadas da câmera.

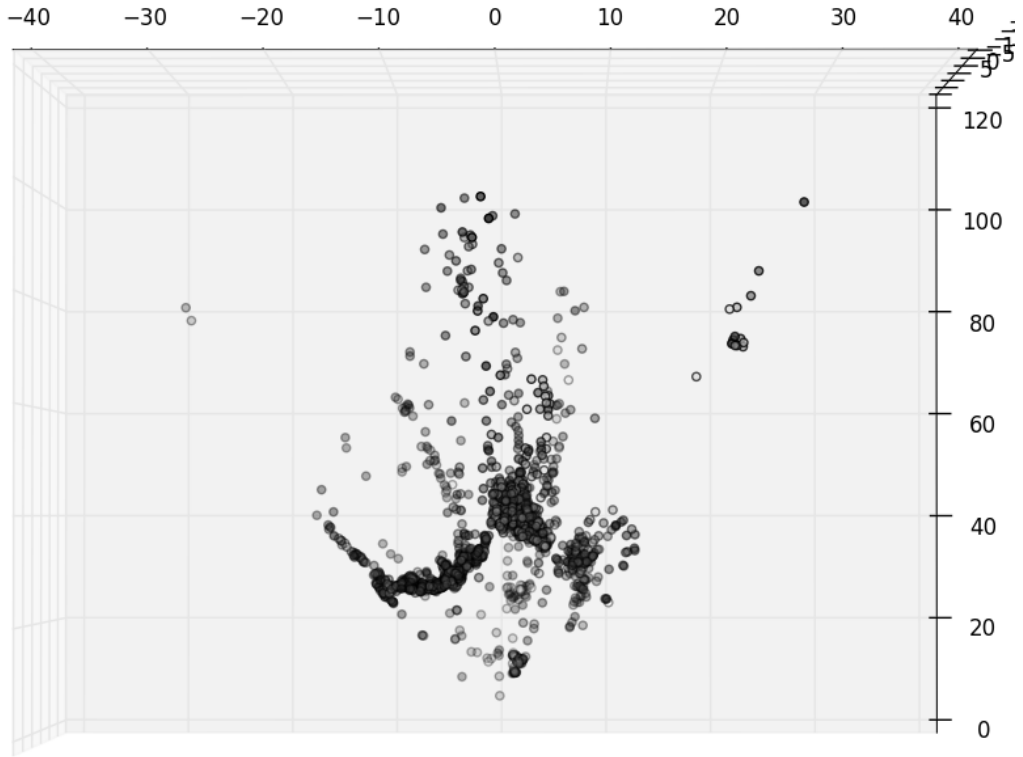
Com base nas informações de translação $\vec{T}$ e rotação $\vec{\Omega}$ utiliza-se o método de Rodrigues (BROCKETT, 1984) para calcular a matriz de transformação da câmera $M_{C,W}$. Desse modo, o cálculo da posição de cada ponto P_W no sistema de coordenadas global torna-se apenas uma multiplicação matricial

Figura 6 – Imagem com profundidades. KITTI - Sequência 3, *frames*: 0 ao 10



Fonte: do autor

Figura 7 – Pontos no sistema de coordenadas da câmera. KITTI - Sequência 3, *frames*: 0 ao 10



Fonte: do autor

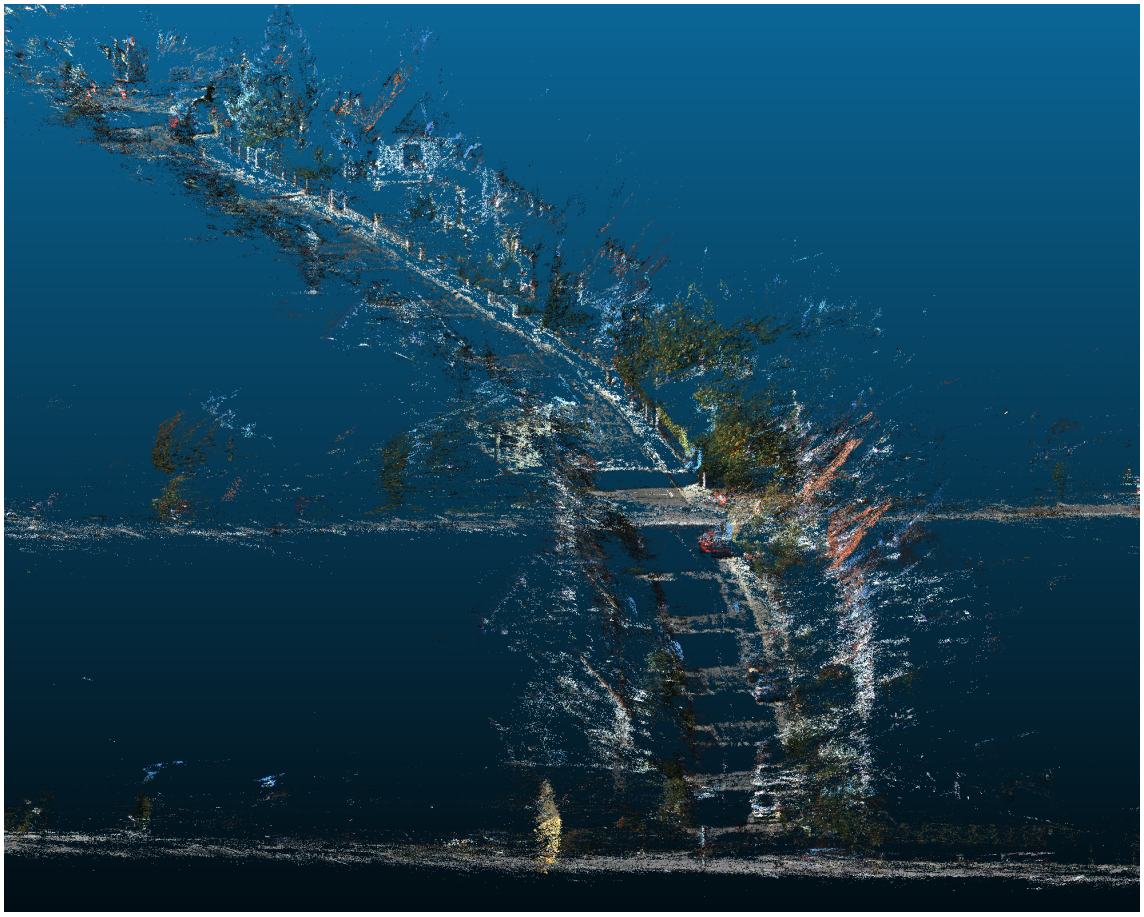
$$P_W = R_{C,W} \times P_C + T_{W,C} \tag{30}$$

Utilizando-se a Equação (30) os pontos são então transpostos para o sistema de coordenadas global (P_W) e concatenados *frame* a *frame*, escolhendo-se os melhores resultados de cada ponto. Também é utilizado um limiar de profundidade, em que, pontos muito distantes, e, portanto, mais suscetíveis a erros, não são considerados para a transposição.

Os pontos no espaço 3D são armazenados em uma estrutura de dados própria, chamada nuvem de pontos. Essa estrutura contém as coordenadas, o identificador do ponto e o erro de cálculo obtido na estimativa. Assim, foi possível gerar um modelo 3D a partir desses pontos. Para as sequências disponíveis foram obtidos os modelos 3D do ambiente, utilizando o algoritmo proposto.

A Figura 8 mostra um exemplo de nuvem de pontos obtido a partir da sequência 5 do KITTI, utilizando todos os *frames* da sequência. É possível distinguir claramente a rua onde o veículo transitou, os carros estacionados, bem como as árvores e algumas estruturas ao redor.

Figura 8 – Exemplo de Nuvem de pontos. KITTI - Sequência 5

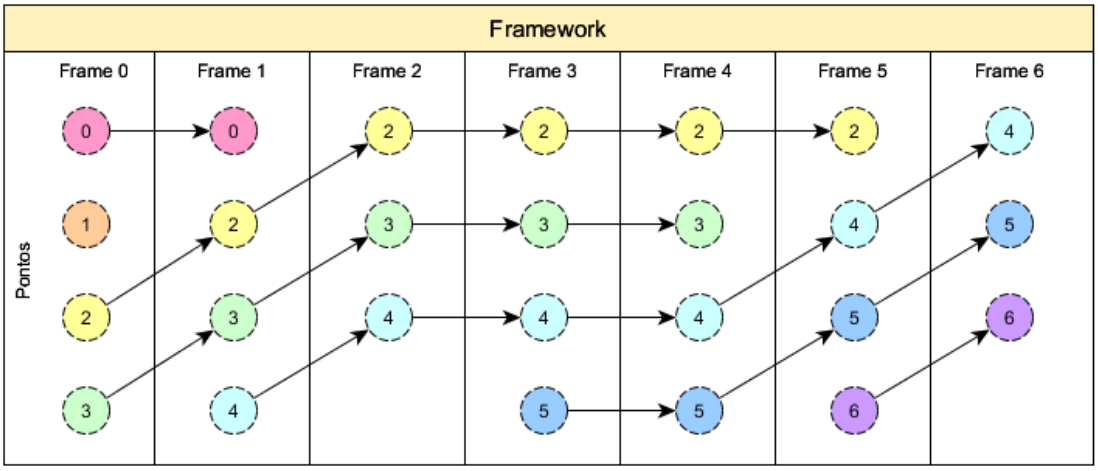


Fonte: do autor

6.5 Estrutura de dados

Para lidar com a grande quantidade de dados que é utilizada, uma estrutura de dados para o tratamento das *features* nos *frames* teve de ser elaborada. A estrutura de dados foi montada em formato de grafo estruturado em treliça, conforme mostrado na Figura 9, sendo capaz de lidar com a grande quantidade de dados a serem processados em uma sessão. Os nodos do grafo representam as *features* extraídas da imagem, e a ligação entre esses nodos é a correspondência da *feature* no próximo *frame*.

Figura 9 – Estrutura de dados das *features* rastreadas



Fonte: do autor

Esses nodos podem ser interpretados pela aplicação de diversas formas, conforme o contexto do processamento. Além disso, seus dados podem ser utilizados em funcionalidades específicas durante o processamento de imagens ou o cálculo geométrico: a ligação entre os nodos pode ser utilizada no cálculo de profundidade, possuem identificação própria, e podem ser referenciados de forma única em qualquer etapa do processamento, bem como podem ser incluídos, excluídos e alterados individualmente ou em bloco.

As localizações das *features* no plano da imagem e sua relação com as *features* do *frame* anterior são adicionadas à estrutura em um novo índice.

6.6 Exportação dos dados

Ao final do processamento, os pontos armazenados são exportados para um arquivo de nuvem de pontos, onde podem ser visualizados em um software de visualização de nuvens de ponto ou no software desenvolvido exclusivamente para esse fim.

6.7 Análise de Erros

Em ambientes reais, a estimativa de profundidade possui um erro que somente pode ser inferido por outras medidas indiretas. Muitos parâmetros têm se mostrados efetivos na determinação de pontos como válidos ou inválidos (TRIGGS, 1996). Não obstante, a determinação de limiares pode ser uma tarefa complexa. Um limiar muito permissivo compromete a efetividade da medida, eis que permite que pontos errôneos sejam interpretados como válidos. Por outro lado, um limiar muito restritivo acaba inutilizando pontos que poderiam contribuir para a construção da cena. Ainda, muitas vezes não há um ponto de separação claro, e a escolha de um valor de limiar significa abrir mão de pontos válidos para remover os indesejados.

De forma a conseguir estimar o impacto da escolha da medida de erro, o dataset do desafio 3D-RMS (TYLECEK *et al.*, 2019) foi utilizado. Esse dataset disponibiliza o *ground truth* das medidas de profundidade nas séries de treino, uma vez que são dados sintéticos. Uma rodada do algoritmo foi executada e os dados intermediários das medidas de profundidade foram armazenados, sem se realizar nenhuma filtragem. Além disso, o valor verdadeiro de profundidade foi armazenado em conjunto com cada *feature*. O erro foi estabelecido como sendo a diferença absoluta ente o valor da profundidade calculada e o valor verdadeiro.

Para interpretar o erro, os pontos foram divididos em quatro grupos, conforme o valor do erro. Os grupos e os valores de limite mínimo e máximo de erro foram determinados empiricamente e podem ser vistos na Tabela 3. No primeiro grupo, encontram-se os pontos cujo erro de medida é inferior a 0,25m. No segundo grupo, os pontos onde o erro está entre 0,25m e 0,5m. No terceiro grupo, os pontos com erro entre 0,5m e 1,0m e no quarto e último grupo os pontos com erro superior a 1m.

Tabela 3 – Grupos de Pontos conforme erro

Nome	Tipo	Mínimo	Máximo
Grupo 1	Válido	0,00m	0,25m
Grupo 2	Depende	0,25m	0,50m
Grupo 3	Depende	0,50m	1,00m
Grupo 4	Inválido	1,00m	-

A divisão dos grupos nesses valores se deu de forma a separar os pontos conforme sua utilidade para compor a nuvem de pontos. O Grupo 1 contém aqueles pontos que foram considerados bons quando da realização pelo desafio 3D-RMS (TYLECEK *et al.*, 2019), que considerava os pontos quando encontrados até aproximadamente o valor de 0,25m. Já os Grupos 2 e 3 contêm os pontos que dependendo da aplicação podem ser considerados bons. Para fim de cálculo de erro, eles foram considerados como pontos que podem ser aproveitados em uma nuvem de pontos que reconstrói um ambiente aberto 3D. No Grupo

4, foram classificados os pontos que são considerados inválidos, pois estando mais do que 1m do valor real já se compromete a construção da nuvem de pontos. Esses números são utilizados para medidas que vão desde centímetros até aproximadamente 20 metros. Dessa forma, o erro de 0,25 metros pode ser considerado baixo para a reconstrução.

6.7.1 Critério de Profundidade Negativa

O primeiro critério para eliminar pontos após o cálculo de profundidade é o próprio valor de profundidade. Valores negativos são geometricamente impossíveis de serem medidos, haja vista que tais pontos estariam atrás da câmera, o que não faz sentido do ponto de vista prático. A Tabela 4 mostra o resultado da filtragem dos pontos negativos em cada um dos grupos. O resultado da filtragem mostra que 35,3% dos pontos inválidos são eliminados, sem muito impacto sobre os pontos válidos. É importante ressaltar que esse critério é simples de implementar, tendo em vista que o custo para a computação é apenas a comparação do resultado, sem necessidade de cálculo extra.

Tabela 4 – Impacto de eliminar valores de profundidade calculados como negativos sobre os pontos, por grupos

Grupo	Sem filtro	$\rho > 0$	
	# pontos	# pontos	dif. (%)
Grupo 1	95011	95011	0,0%
Grupo 2	85900	85896	0,0%
Grupo 3	78494	78216	-0,4%
Grupo 4	171975	111277	-35,3%

6.7.2 Critério de Erro de Reprojeção

Um dos métodos mais utilizados para determinar o erro da medida de profundidade é o erro de reprojeção (VEDALDI; GUIDI; SOATTO, 2007; LIM; LIM; KIM, 2014; PEREIRA, 2018). Conforme explicado na Seção 2.1.6, essa medida estima a posição de um ponto no sistema de coordenadas global a partir dos dados de imagem (x, y, ρ) para, em seguida, estimar a posição no plano de imagem em outra posição de câmera. Essa posição é comparada com a posição encontrada para o ponto por uma outra medida – que nesse trabalho foi realizada através do rastreamento de *features*. A diferença em pixels entre as medidas determina esse erro. A Tabela 5 mostra o impacto dessa medida nos pontos em cada grupo. Os pontos usados como referência são aqueles já removidos com a estimativa de profundidade negativa.

A Tabela 5 mostra que a filtragem pelo erro de reprojeção tem efetividade limitada. Por exemplo, tolerar um valor de até 50 pixels de erro elimina 34,6% de pontos do grupo 4, abrindo mão de menos de 15% dos demais grupos. Por outro lado, limitar o erro a no máximo 10 pixels acaba eliminando uma grande quantidade de pontos dos grupos 1 e 2

Tabela 5 – Grupos de Pontos após filtragem por diferentes limiares no Erro de Reprojeção

Grupo	$\rho > 0$	Err Reproj < 50		Err Reproj < 25		Err Reproj < 10	
	# pontos	# pontos	dif. (%)	# pontos	dif. (%)	# pontos	dif. (%)
Grupo 1	95011	90611	-4,6%	85873	-9,6%	71279	-25,0%
Grupo 2	85896	80998	-5,7%	76747	-10,7%	58009	-32,5%
Grupo 3	78216	66554	-14,9%	57793	-26,1%	36590	-53,2%
Grupo 4	111277	72734	-34,6%	54878	-50,7%	23796	-78,6%

(25% e 32,5%, respectivamente), e possui grande efetividade sobre os pontos do grupo 4, eliminando 78,6% dos pontos desse grupo. Um valor intermediário pode trazer uma filtragem razoável por um custo menor de pontos bons sendo eliminados.

6.7.3 Critério de Erro de Reprojeção em Conjunto com a Profundidade

Um método alternativo para se medir o erro é proposto. Consiste em combinar a medida de erro de reprojeção e a distância do ponto. A ideia por trás da medida é que pontos mais afastados tendem a possuir menor erro de reprojeção, mesmo que sua medida tenha um erro maior, devido ao efeito de paralaxe. Para se compensar esse fenômeno, se testou dividir o valor de erro de reprojeção pelo valor de ρ e aplicar um limiar nesse valor. O resultado pode ser visto na Tabela 6.

Tabela 6 – Grupos de Pontos após filtragem por diferentes limiares no Erro de Reprojeção$/\rho$

Grupo	$\rho > 0$	Err Reproj$/\rho < 50$		Err Reproj$/\rho < 25$		Err Reproj$/\rho < 10$	
	# pontos	# pontos	dif. (%)	# pontos	dif. (%)	# pontos	dif. (%)
Grupo 1	95011	86911	-8,5%	76677	-19,3%	39767	-58,1%
Grupo 2	85896	77938	-9,3%	63735	-25,8%	20030	-76,7%
Grupo 3	78216	60656	-22,5%	36737	-53,0%	9414	-88,0%
Grupo 4	111277	58003	-47,9%	31409	-71,8%	12766	-88,5%

Nota-se, pelos valores da Tabela 6 que a medida proposta é bastante agressiva. Tanto os valores do grupo 4, quando os valores dos outros grupos são bastante afetados por um limiar simples sobre o valor. Entretanto, nota-se sensibilidade dos pontos considerados inválidos a um limiar da medida. Um limiar em valor de 50 é capaz de filtrar 47,9% dos pontos do grupo 4, enquanto um valor de 10 filtra 88,5% desse mesmo grupo, sob pena de filtrar 58,1% dos pontos do grupo 1.

6.7.4 Critério de Erro de Diferença de Ângulo de Paralaxe

Por fim, foi testada a sensibilidade da medida proposta na Subseção 5.3.2. Os valores dos pontos divididos por grupos após a filtragem pelos valores γ_μ e γ_σ são mostrados nas

tabelas 7 e 8, respectivamente.

Tabela 7 – Grupos de Pontos após filtragem por diferentes limiares em γ_μ

Grupo	$\rho > 0$	$\gamma_\mu < 0,5$		$\gamma_\mu < 0,25$		$\gamma_\mu < 0,15$	
	# pontos	# pontos	dif. (%)	# pontos	dif. (%)	# pontos	dif. (%)
Grupo 1	95011	86613	-8,8%	77512	-18,4%	69360	-27,0%
Grupo 2	85896	77312	-10,0%	68764	-19,9%	61467	-28,4%
Grupo 3	78216	64278	-17,8%	51458	-34,2%	42648	-45,5%
Grupo 4	111277	70561	-36,6%	47930	-56,9%	36000	-67,6%

Tabela 8 – Grupos de Pontos após filtragem por diferentes limiares em γ_σ

Grupo	$\rho > 0$	$\gamma_\sigma < 1,5$		$\gamma_\sigma < 1,25$		$\gamma_\sigma < 1,0$	
	# pontos	# pontos	dif. (%)	# pontos	dif. (%)	# pontos	dif. (%)
Grupo 1	95011	93476	-1,6%	90636	-4,6%	85742	-9,8%
Grupo 2	85896	84282	-1,9%	81707	-4,9%	76758	-10,6%
Grupo 3	78216	75936	-2,9%	72037	-7,9%	66376	-15,1%
Grupo 4	111277	104603	-6,0%	95654	-14,0%	82588	-25,8%

Assim como nas outras medidas, essas medidas não são capazes de separar inequivocamente os pontos entre os grupos por um simples limiar.

6.7.5 Filtragem Multicritérios

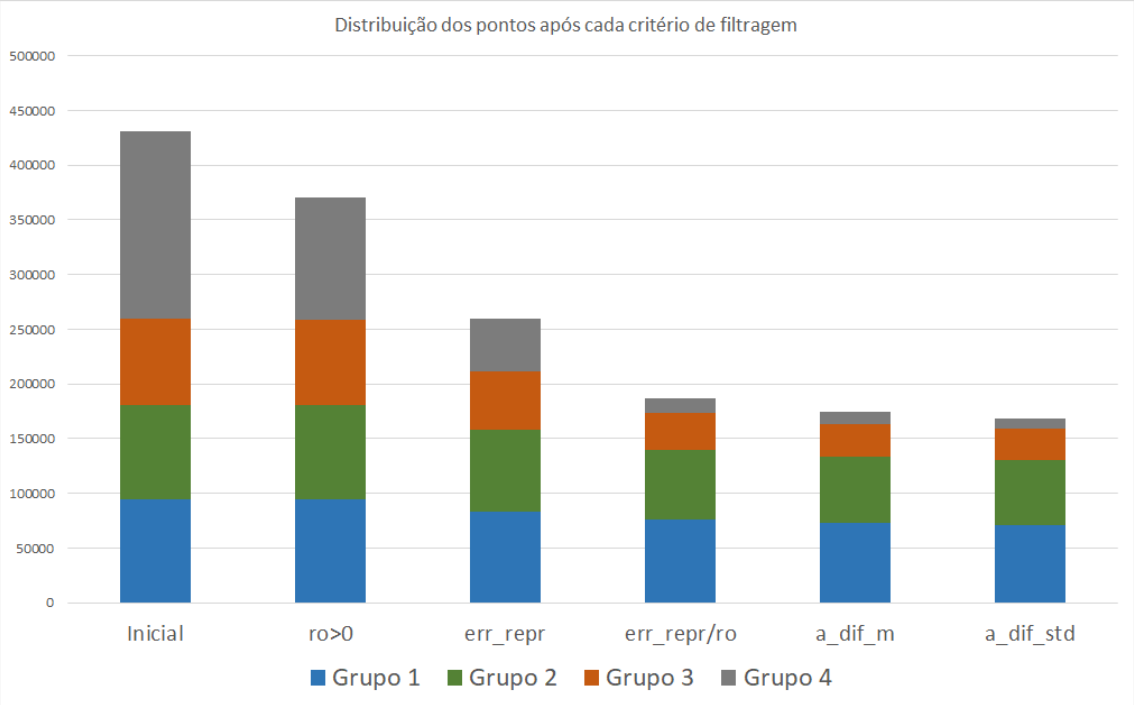
Para realizar a filtragem de pontos foi desenvolvido um filtro que contemplava limiares para todas essas medidas exploradas nessa seção. Seus valores foram determinados empiricamente e o resultado pode ser observado na Tabela 9.

Tabela 9 – Grupos de Pontos após sucessivas filtragens por limiares de ρ, Erro de Reprojeção, Erro de Reprojeção$/\rho$, γ_μ e γ_σ

Grupo	Sem Filtro	$\rho > 0$	Erro Reproj < 20	Erro Reproj$/\rho < 25$	$\gamma_\mu < 0,25$	$\gamma_\sigma < 1,25$
Grupo 1	95011	95011	83793	76498	73085	71486
Grupo 2	85900	85896	74016	63222	60028	58774
Grupo 3	78494	78216	53478	33364	30549	29282
Grupo 4	171975	111277	48042	13632	10374	9117

Com a adoção desses critérios, foi possível diminuir os pontos do Grupo 4, considerados inválidos, em até 94,7%. A perda no Grupo 1 e 2 foram de 24,8% e 31,6%, respectivamente. Já o Grupo 3 teve diminuição de 62,7% no número de pontos. O resultado combinado dos critérios foi superior ao resultado individual de cada critério. O impacto na relação entre os pontos de cada grupo pode ser visto na Figura 10.

Figura 10 – Distribuição dos pontos pelos grupos após cada critério de filtragem



Fonte: do autor

É possível notar que os pontos do Grupo 4 representavam uma grande porcentagem do total de pontos, atingindo 39% dos pontos iniciais. Após a filtragem, tais pontos foram reduzidos para 5,4% do total de pontos. Já os pontos dos Grupos 1 e 2 tiveram baixo impacto de perda de pontos, mas aumentaram sua participação drasticamente, passando de 22% para 42% no caso do Grupo 1 e 20% para 35% no caso do Grupo 2. Já os pontos pertencentes ao Grupo 3 mantiveram a participação, oscilando de 18% para 17%.

6.8 Resultados

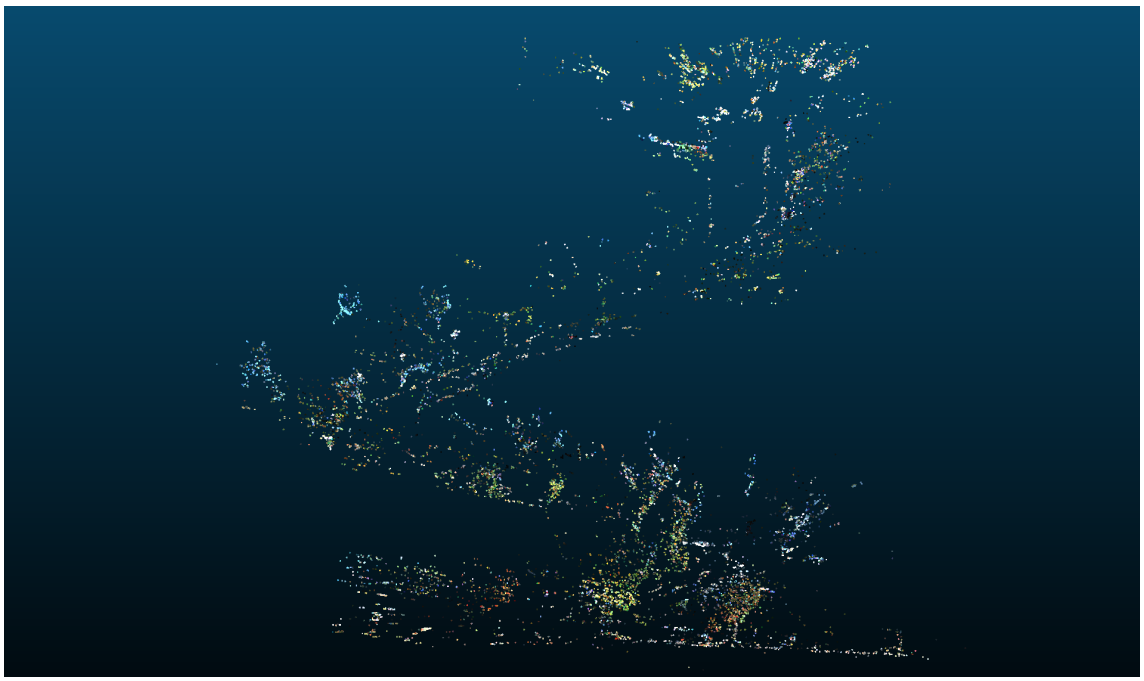
A solução proposta possui resultados relevantes, gerando nuvens de pontos com alta qualidade visual. Para configurar o algoritmo, pode-se escolher dois principais parâmetros: a quantidade máxima de pontos por *frame* que serão rastreados e a quantidade de *frames* que serão considerados para realizar o cálculo de profundidade. A quantidade de pontos é diretamente proporcional ao tempo de execução, uma vez que cada etapa do algoritmo será executada sobre cada ponto que foi rastreado, entretanto, poucos pontos pode tornar o resultado muito esparso, comprometendo sua qualidade, portanto uma solução de compromisso deve ser buscada, conforme o poder computacional e a qualidade desejada. Já a quantidade de *frames* considerados para realizar o cálculo de profundidade impacta na qualidade do dado a ser gerado. Quantos mais *frames* forem considerados para o cálculo de profundidade, menos impacto no resultado final tem uma medida errada de um ponto. Entretanto, há um custo de menos pontos terem sido rastreados por tantos

frames consecutivos. Isso afeta a quantidade final de pontos calculados. Novamente, uma solução de compromisso deve ser buscada. Já os demais parâmetros intermediários foram utilizados nos valores padrão de cada algoritmo.

O número de *frames* para realizar o cálculo de profundidade foi estimado dinamicamente para o valor que corresponde ao movimento da câmera equivalente a 3 metros. Esse valor foi determinado empiricamente como mais adequado para o dataset do KITTI por apresentar o melhor compromisso entre a quantidade mínima de amostras e a velocidade de cálculo.

A quantidade máxima de pontos por *frame* a serem rastreados deve ser inicializado no início do algoritmo e tem grande influência no tempo de execução. Por isso, diversos valores foram testados e o resultado prático de tempo é dependente com a plataforma. Como forma de exemplo, três opções foram executadas e os resultados visuais podem ser comparados. A primeira, mostrada na Figura 11, 2048 pontos por *frame* foram utilizados. O algoritmo rodou em aproximadamente 35s para os 800 *frames* no computador utilizado para os testes. A segunda, mostrada na Figura 12, 6000 pontos por *frame* foram utilizados. O algoritmo rodou em aproximadamente 80s, para a mesma quantidade de *frames* no mesmo computador. Já na terceira, mostrada na Figura 13, 250000 pontos por *frame* foram utilizados. O algoritmo rodou em aproximadamente 2500s.

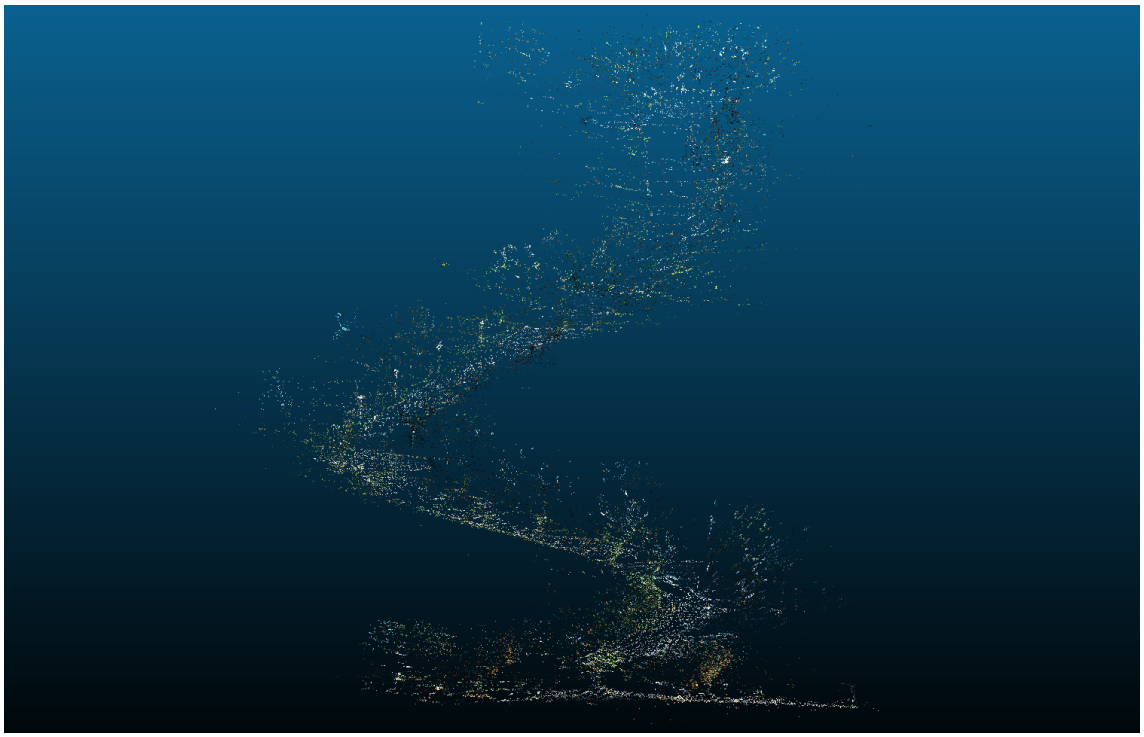
Figura 11 – Nuvem de pontos gerada com limite de 2048 *features* rastreados por *frame*. KITTI - Sequência 3



Fonte: do autor

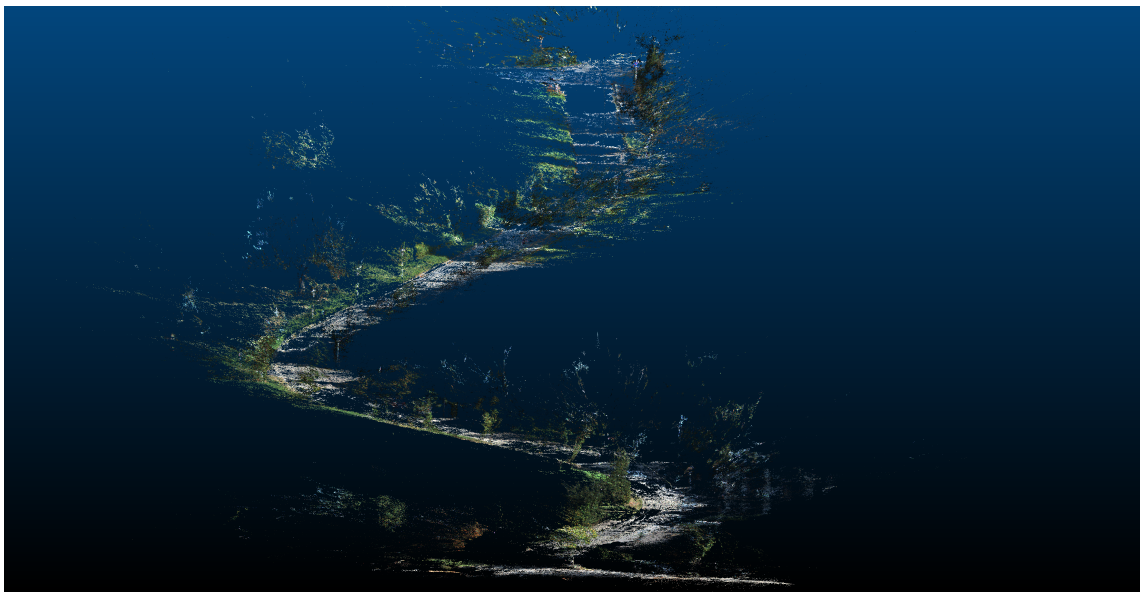
Pode-se notar claramente uma evolução na qualidade da nuvem de pontos gerada, em contrapartida, mais tempo é necessário para se realizar o cálculo necessário. Como o cálculo é realizado para cada ponto, individualmente, o tempo de execução é linearmente

Figura 12 – Nuvem de pontos gerada com limite de 6000 *features* rastreados por *frame*. KITTI - Sequência 3



Fonte: do autor

Figura 13 – Nuvem de pontos gerada com limite de 250000 *features* rastreados por *frame*. KITTI - Sequência 3



Fonte: do autor

dependente à quantidade de pontos. Isso torna o algoritmo altamente escalável, uma vez que um crescimento linear de dados na entrada, acarreta numa necessidade linear de mais recursos de processamento e de memória. O desempenho do algoritmo é melhor discutido na Seção 6.9.

6.8.1 Outros Resultados

Durante a construção dessa nuvem de pontos, utilizando os dados já obtidos é possível realizar cálculos geométricos para, por exemplo, realizar a separação de objetos, como, edificações, vegetação e outros veículos. Outro exemplo é a possibilidade de, ainda, realizar a separação do polígono que compõe a pista de rodagem, para extrair informações importantes ao veículo que nela transita.

Alguns testes já foram realizados para a obtenção de planos a partir desses pontos. Utilizando o método de RANSAC, em conjunto com o modelo tridimensional para planos, foi possível estabelecer os planos principais das nuvens de pontos, como por exemplo a rua.

Os pontos que foram identificados no *frame* atual como pertencentes a algum plano são utilizados como semente para um processo de segmentação na imagem. Esse processo de segmentação tem como objetivo identificar outros pontos da imagem que sejam pertencentes ao mesmo plano.

Os pontos encontrados na imagem como sendo pertencentes ao plano são então projetados do plano da imagem para o plano no espaço utilizando a técnica de projeção de uma linha em um plano. Ambos são multiplicados pela matriz de rotação da câmera e convertidos ao espaço 3D, onde está definido o plano.

É feito então o cálculo do ponto de cruzamento da linha no plano. O ponto resultante é então incluído na nuvem de pontos.

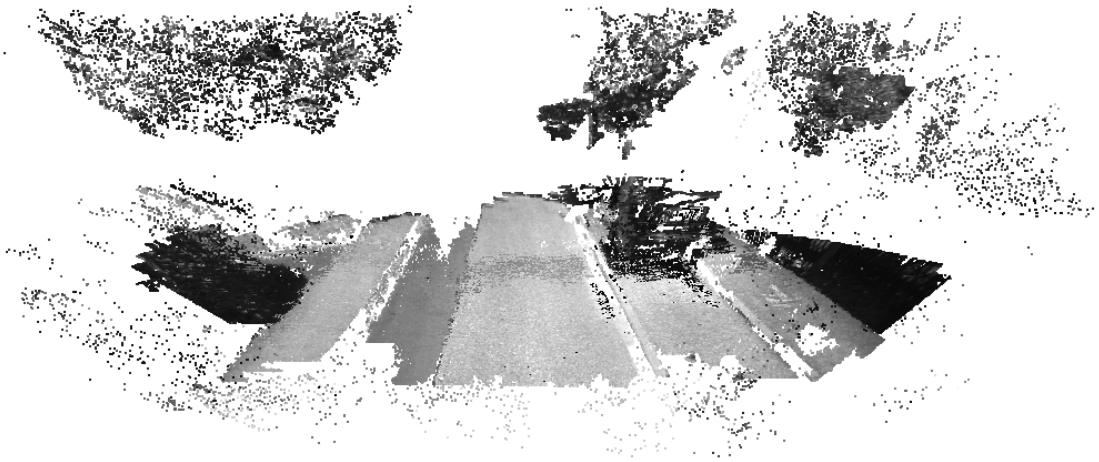
A Figura 14 mostra o resultado de um desses testes, com um algoritmo de segmentação utilizando a média e o desvio padrão dos tons de cinza dos pontos do plano como referência para a segmentação. Comparando com a Figura 15, que traz um dos *frames* utilizados no cálculo, o resultado é visualmente muito promissor, mas ainda precisa de alguns ajustes.

6.8.2 3D Reconstruction meets Semantics 2018 – Challenge

De forma a testar o método e comparar com outras soluções de reconstrução 3D, o modelo proposto foi submetido ao Desafio organizado no ECCV 2018 Workshop (TYLECEK *et al.*, 2019). Nesse desafio, um robô com 5 pares de câmeras estéreo foi movimentado em um jardim e suas posições foram medidas com auxílio de equipamentos externos. O objetivo do desafio era reconstruir o ambiente utilizando as imagens das câmeras, dado as posições e orientações de cada imagem.

O dataset era dividido em três partes: uma de treino, gerada artificialmente, onde era

Figura 14 – Nuvem de pontos com plano segmentado. KITTI - Sequência 3, *frames* 0 ao 24



Fonte: do autor

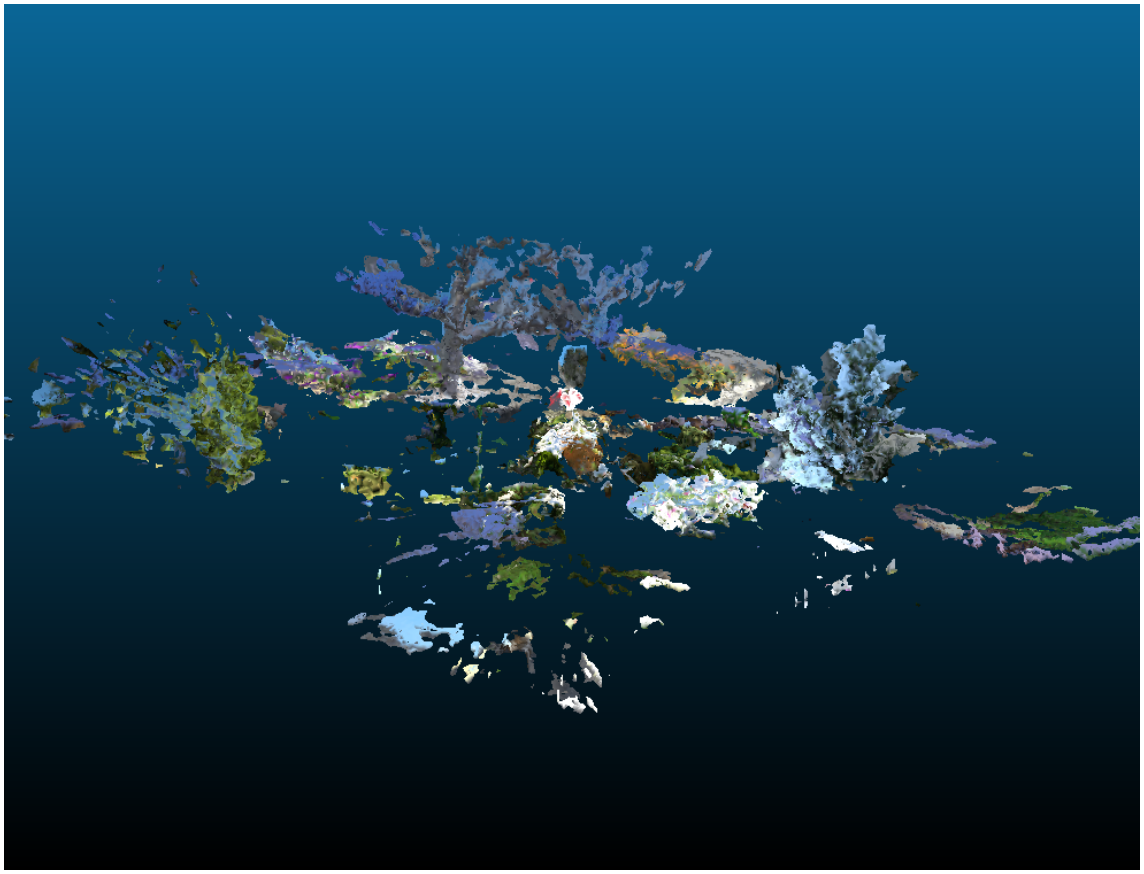
Figura 15 – KITTI - Sequência 3. *Frame* 18

Fonte: (GEIGER; LENZ; URTASUN, 2012)

fornecido o *ground truth* tanto dos valores de profundidade dos pixels quanto do ambiente 3D, para se desenvolver o algoritmo; uma de testes, gerada artificialmente, onde o *ground truth* também era fornecido, para se testar a solução; e uma de validação, contendo somente as imagens do robô, tiradas de um ambiente real.

Duas versões do algoritmo foram geradas e rodadas sobre o dataset do desafio. A versão *lapsi360* apresenta o algoritmo rodando sobre todas as câmeras, enquanto *lapsi4* utilizou apenas quatro câmeras, as duas da frente e duas na lateral. Se optou por essa segunda configuração por se assemelhar mais com a disposição do robô real do desafio, que possuía apenas as quatro câmeras. Os resultados deveriam ser encaminhados como uma mesh 3D conectando os pontos encontrados. Para calculá-la a partir da nuvem de pontos, foi utilizado o aplicativo CloudCompare (GIRARDEAU-MONTAUT, 2011). O resultado da reconstrução do dataset de teste pode ser visualizado na Figura 16.

Figura 16 – Resultado do algoritmo no 3D RMS 2018. 3D mesh gerada no CloudCompare a partir do resultado do *lapsi4* rodando na sequência de teste



Fonte: do autor

Os resultados são apresentados na Figura 17. Na primeira linha são mostrados os resultados no dataset de teste, enquanto na segunda, os resultados no dataset de validação (real). A primeira coluna traz os resultados de precisão, enquanto a segunda os resultados completude. As outras soluções propostas são mostradas como HAB e DTIS. Colmap é um software de reconstrução 3D offline que foi utilizado como referência pelos organiza-

dores do evento.

Os resultados mostram que ambas as soluções apresentadas tiveram precisão muito boa nos dois cenários. Os resultados do *lapsi4* (0,166m) e do *lapsi360* (0,164m) no cenário de testes acabaram não ficando melhor que os dos outros concorrentes. Entretanto, nos dados reais, os resultados do *lapsi4* e *lapsi360* (0,150m em ambos) só não foram melhores que o do Colmap, que foi utilizado como referência. Por se tratar de um algoritmo de reconstrução semi-denso, era de se esperar que a completude ficasse abaixo dos demais. Isso pode ser notado no ambiente de teste, onde o valor de 23,9% do *lapsi360* é bastante inferior aos demais. Contudo, no ambiente real a performance de todos os algoritmos caíram bastante. Dessa forma, a diferença entre os 13,7% do *lapsi360* pode ser comparada com 27,1% do DTIS e até mesmo com o 35,8% do Colmap, um dos mais desenvolvidos softwares em reconstrução 3D offline.

6.9 Desempenho

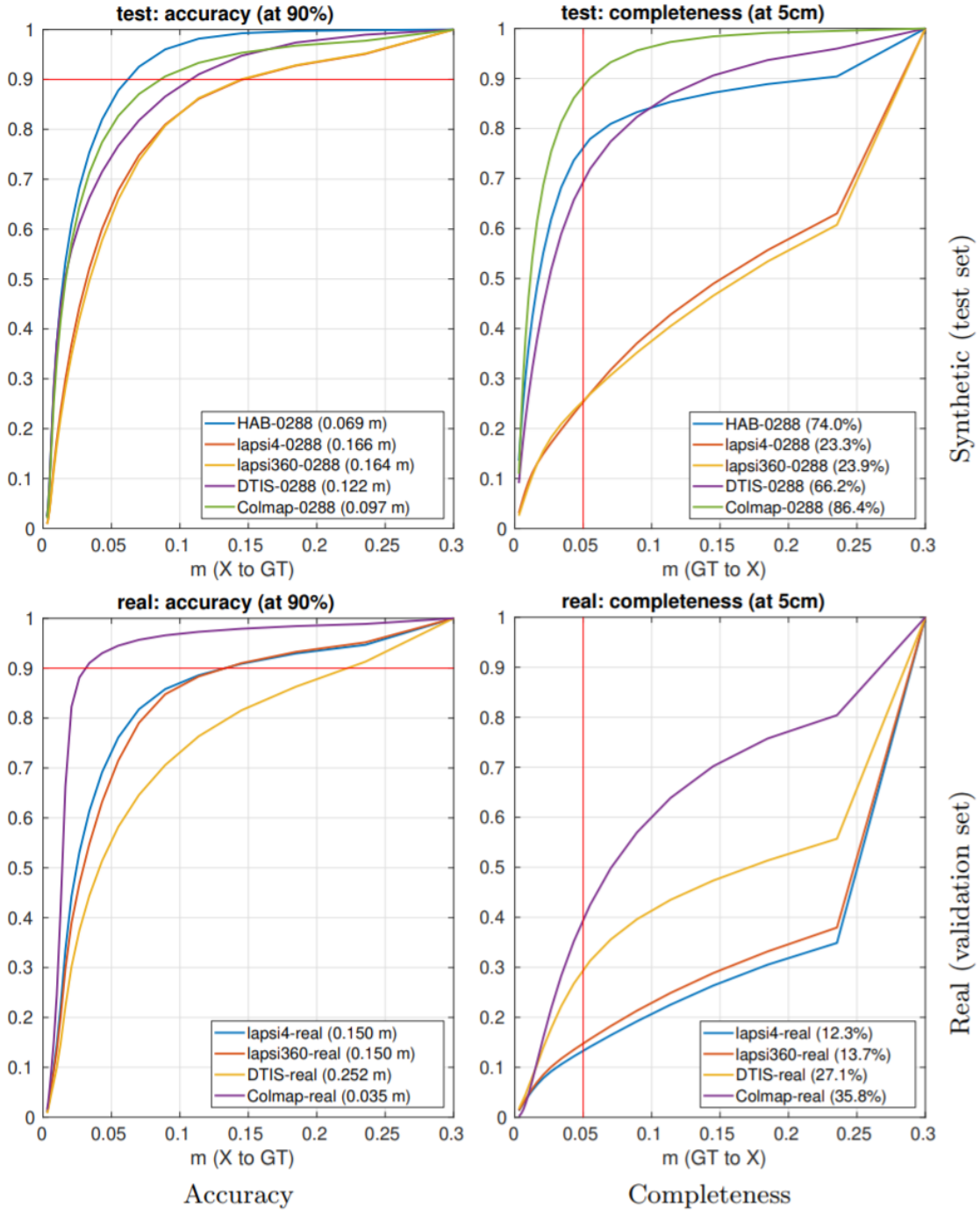
Para definir em quais módulos é possível obter maior ganho em performance, as etapas de implementação e testes foram seguidas por etapas de *profiling*. O resultado do *profiling* do software é mostrado na Tabela 10. Nela são mostrados os tempos de execução do algoritmo rodando no dataset do KITTI, na sequência 3. A função de escolha de *features* foi limitada de forma que cada *frame* possuía no máximo 5000 pontos a serem rastreados. Nessa configuração, o tempo para calcular os pontos de cada *frame* é de 85ms em média, contra o tempo de 100ms por *frame* do dataset.

Para se ter uma ideia do esforço computacional de cada função, as dimensões dos dados que cada uma deve operar foram obtidos do algoritmo. No exemplo apresentado, o algoritmo de Lucas-Kanade operou em média sobre 4875 *features* por *frame*. O cálculo de profundidade, sobre 3334 pontos em cada *frame*. Desses, em média, 618 foram considerados válidos e incluídos na nuvem de pontos, passando pela última etapa de transformação do sistema de coordenadas da câmera para o sistema global em uso.

Na Tabela 10 pode-se notar que a fase de rastreamento é a que mais consome tempo de processamento, seguido pela fase de geração de novas *features*. Isso está de acordo com o esperado, pois essas funções são as que possuem maior complexidade computacional.

A Figura 18 mostra a divisão percentual do tempo de cada função. A parte do algoritmo responsável pelo rastreamento das *features*– Lucas-Kanade – executa durante 55,66% do total do tempo, consumindo boa parte do processamento do algoritmo. Em seguida, o algoritmo de escolha das *features*– GoodFeaturesToTrack – executa durante 16,85% do tempo. O tempo gasto para buscar as imagens no disco – GetImg – demora 13,24% do tempo e o cálculo da profundidade dos pontos – GetDepth – consome 7,60% do tempo. Esses dados mostram que o algoritmo que possui maior gasto computacional é o rastreamento de *features*. Isso se deve ao principalmente ao fato de que o algoritmo

Figura 17 – Comparativo dos resultados do algoritmo no 3D RMS 2018



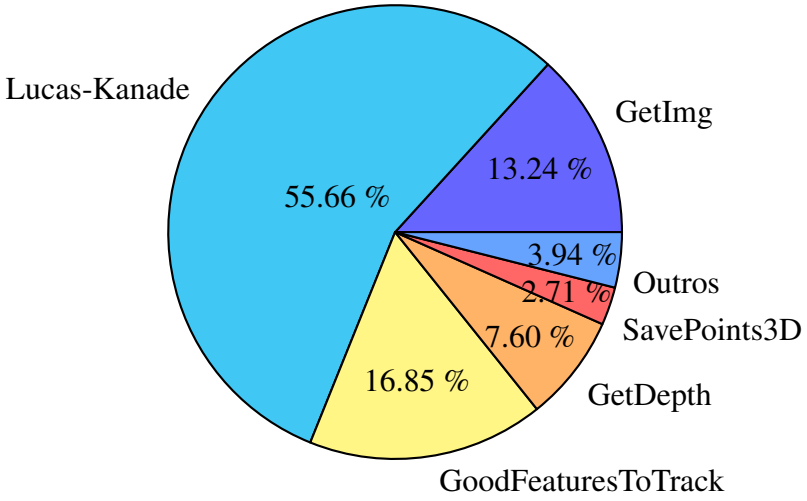
Fonte: (TYLECEK *et al.*, 2019)

Tabela 10 – Desempenho das funções com maior tempo em execução. Dataset: KITTI, Sequência: 03, limite de 5000 pontos por *frame*

Função	Descrição	Repet.	Pontos /repet.	Tempo (s)
GetImg	Busca próximo *frame*	801	-	9,92
Lucas-Kanade	Rastreamento de *features*	800	4875	41,69
GoodFeaturesToTrack	Escolha de *features*	801	-	12,62
GetDepth	Cálculo de profundidade	796	3334	5,69
SavePoints3D	Conversão ao espaço 3D	796	618	2,03
Outros				2,95
Total				**74,90**

opera sobre todas as *features* escolhidas para serem rastreadas, enquanto os outros algoritmos operam em frações de pontos, diminuindo sua carga.

Figura 18 – Resultados de *Profiling*



Fonte: do autor

Dos resultados de *profiling* é possível obter dados de quais funções são críticas ao desempenho do algoritmo e devem ser otimizadas a cada etapa. Também é uma ferramenta importante para determinar se uma solução deve ou não ser utilizada, pois um impacto significativo no desempenho pode torná-la uma boa opção ou inviabilizá-la, dependendo do caso.

6.10 Conclusão do Capítulo

Nesse capítulo foi mostrado um método para cálculo de profundidade de pontos em uma imagem com complexidade computacional compatível para ser executada em qualquer plataforma. Um algoritmo para reconstrução 3D foi apresentado buscando otimizar o cálculo, obtendo o melhor resultado possível dentro das restrições impostas a um pro-

grama que se deseja embarcar. No próximo capítulo, uma arquitetura computacional embarcada é proposta para dar suporte à execução de algoritmos de visão computacional.

7 MPVUE - MPSOC PARA VISÃO COMPUTACIONAL

Nesse capítulo é apresentado o MPVue, uma arquitetura de hardware para MPSoCs, heterogênea e de memória distribuída, que busca atender o campo da visão computacional. O sistema foi construído buscando o compromisso entre alto desempenho necessário para os algoritmos de visão computacional com o baixo consumo desejado em sistemas embarcados. Para se atingir isso, uma rede em chip em malha foi construída conectando diversos processadores do tipo RISC-V.

O presente capítulo é dividido da seguinte forma: a Seção 7.1 explica brevemente os conceitos da arquitetura de hardware; a Seção 7.2 traz os detalhes da rede em chip proposta no trabalho; a Seção 7.3 apresenta a arquitetura dos elementos processadores do sistema; enquanto a Seção 7.4 mostra os testes que foram realizados na plataforma.

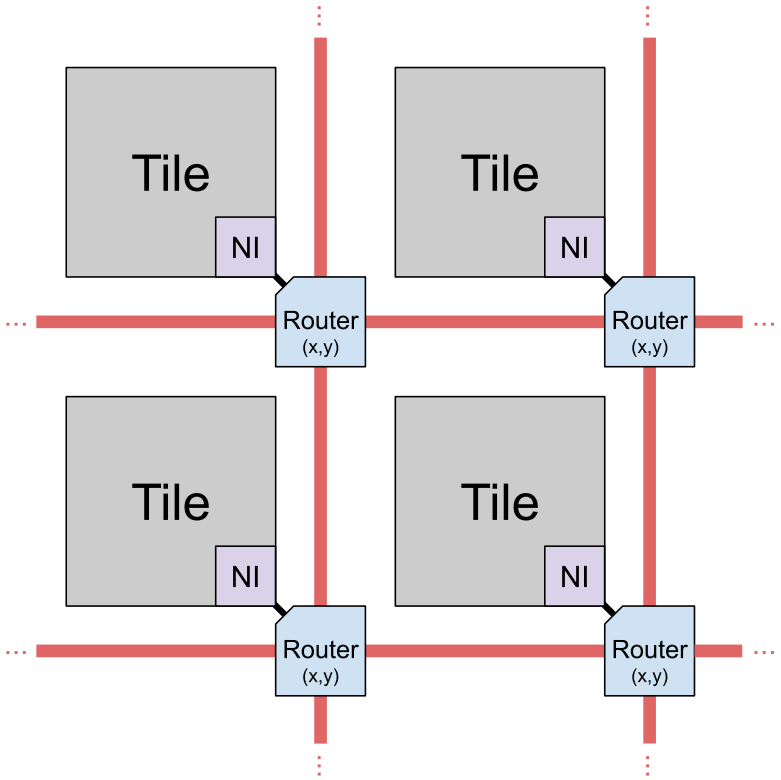
7.1 Arquitetura de Hardware

Nessa seção é apresentado o MPVue, uma plataforma para visão computacional que foi desenvolvida para embarcar as soluções de alta performance para processamento de imagens em ambientes com limitação de energia. Seus elementos processadores são plenamente conectados através de uma rede em chip com topologia em malha 2D como pode ser vista na Figura 19. Cada elemento da rede é chamado de *tile*, mostrado em cinza, e pode ser adicionado ou removido do sistema durante a síntese. Cada *tile* normalmente contém um processador e seus periféricos, e em conjunto com o *roteador* ao qual está ligado, mostrado em azul, forma o bloco básico do sistema. A interface de rede (NI – em roxo) é o elemento de ligação entre o *tile* e o *roteador*.

O sistema utiliza uma arquitetura de memória distribuída, ou seja, cada *tile* possui sua própria memória privada com endereçamento local. O sistema oferece ainda, se desejado, suporte ao uso de *tile* de memória compartilhada que pode ser conectada diretamente na rede, disponibilizando capacidade de armazenamento compartilhado.

O sistema é um MPSoC (Sistema em Chip Multi-Processado) heterogêneo que apresenta dois tipos de *tiles* com sua arquitetura: LaPSIman e LaPSIno. O LaPSIman é o componente mais complexo do sistema, já que possui diversos periféricos e uma maior

Figura 19 – Topologia em malha XY da rede em chip



Fonte: do autor

capacidade de memória. O LaPSIno é uma versão reduzida do LaPSIman, possuindo menos memória e apenas um periférico, a interface de rede.

O sistema pode ser facilmente estendido, incluindo ou removendo elementos ou mudando sua organização interna. A arquitetura é genérica o suficiente para rodar qualquer software paralelo. Ainda assim, o MPVue é focado em aplicações de Visão Computacional.

7.2 Rede em Chip

A rede em chip é uma arquitetura que busca solucionar o grande gargalo que há quando múltiplos processadores precisam se comunicar utilizando um ou mais barramentos dentro de um mesmo chip. Dessa forma, a NoC surge como uma alternativa que alia alto desempenho com baixo consumo, sem abrir mão da baixa latência. A rede em chip utilizada nesse projeto foi proposta por (REINBRECHT *et al.*, 2016).

Os dados dentro de uma rede são transmitidos em pacotes. Cada pacote é dividido em *flits*, que é a menor parte indivisível de um pacote. No MPVue, os pacotes são divididos em *flits* de 32-bits que são roteados através de um caminho pela rede até seu destino.

A estrutura da rede em chip, que possui topologia do tipo malha 2D, é estruturada para poder crescer linearmente conforme a necessidade de ligar mais *tiles* ao sistema. A latência é proporcional ao número de saltos na rede que o dado tem que dar até chegar ao seu

destino, e cresce linearmente com a inclusão de *tiles*. Ademais, a arquitetura permite que comunicações simultâneas sejam transmitidas pela rede. Por fim, com a estrutura em malha, é possível desenvolver estruturas não retangulares, contendo um número arbitrário de itens, sem prejudicar o funcionamento do sistema. Por esses motivos, é possível concluir que o sistema de hardware é altamente escalável, pois o aumento linear da quantidade de *tiles* necessita um aumento linear de recursos disponíveis para uso do sistema.

7.2.1 Roteadores

A rede em chip desenvolvida é composta por um *roteador* de cinco portas, conforme mostrado na Figura 19, que são conectados em uma topologia em formato de malha. O roteamento de pacotes em cada *roteador* é feito utilizando o algoritmo de XY – onde pacotes são roteados em preferência na direção X do que na direção Y. Em cada *roteador*, um *crossbar* seleciona a porta a ser utilizada com base em um arbitragem do tipo *round robin*. Dessa forma, os acessos podem ser multiplexados à mesma saída.

Buffers de oito *flits* são utilizados nas portas de entrada, resultando em uma latência de dois ciclos de clock por *roteador*. Múltiplas comunicações entre *tiles* podem ser realizadas ao mesmo tempo, somente limitadas ao tamanho dos buffers e à latência adicionada pela arbitragem nos *roteadores*. Além disso, cada porta possui canal de entrada e de saída, o que contribui para aumentar a taxa de transferência total da rede.

7.2.2 Interface de Rede

A ligação entre um *tile* e a rede é feita através de uma interface de rede, conforme representado na Figura 19. Cada *tile* possui seu endereço único na rede que representa a coordenada (x,y) de seu *roteador* na malha.

Quando dados precisam ser enviados, a interface prepara um pacote. O primeiro *flit* do pacote é chamado de cabeçalho e contém o endereço de destino, o endereço de origem, o comprimento dos dados e outras informações de controle. Em seguida os dados a serem enviados são convertidos à *flits* e adicionados no pacote. Por fim, o último *flit*, chamado de cauda é adicionado, indicando o fim do pacote. O pacote é, então, enviado para a rede através do *roteador* ao qual a interface está conectada (BINESH, 2013).

Ao chegar ao destino, o pacote é aberto pela interface de rede: o cabeçalho é retirado e seus dados são armazenados em uma *fifo*, onde ficam disponíveis ao processador para serem lidos.

Tanto o canal de entrada da interface de rede, quanto o canal de saída possuem *fifo* para armazenar os dados, com capacidade de armazenar 256 *flits*. O tamanho da *fifo* do canal de saída impõe o tamanho máximo do pacote, uma vez que a interface de rede irá adicionar os dados que estiverem na *fifo* no momento de criar o pacote.

A informação de cabeçalho é sobrescrita cada vez que um pacote é recebido. Assim, apenas um pacote pode ser recebido por vez para que não se percam dados. Para evitar a

perda de dados, um semáforo foi implementado em cada interface, de forma a bloquear o recebimento de novos pacotes, antes que o pacote atual tenha sido processado.

7.3 Configurações dos *tiles*

O MPVue é composto por dois elementos básicos: o LaPSIman e os LaPSInos, desenvolvidos no contexto do projeto MPVue. O LaPSIman é o gerente do sistema, é responsável pelo controle do fluxo de entrada e saída de dados, interpretação de comandos e controle dos demais elementos. Já os LaPSInos correspondem aos operários do sistema, são processadores otimizados, podendo conter aceleradores.

Cada *core* do sistema foi implementado utilizando como base a plataforma PULPino (TRABER *et al.*, 2016). Essa plataforma é uma solução completa para desenvolvimento de sistemas em chip. Uma vez que compreende código RTL puro aberto, qualquer ajuste pode ser realizado sem a necessidade de interações com os desenvolvedores originais. Além disso, a plataforma dispõe de uma biblioteca de periféricos disponíveis para o uso conforme a necessidade.

Assim, o *core* RI5CY[1] (TRABER; GAUTSCHI; SCHIAVONE, 2016) foi escolhido e sua arquitetura de processador é compartilhada entre os *tiles* do MPVue. A extensão de cálculo de ponto flutuante de precisão simples foi desabilitada para reduzir a área necessária para a implementação. A estrutura de barramento, a organização de memória e os periféricos básicos também foram herdados da plataforma PULPino.

O uso da mesma arquitetura, mesmo que com diferentes periféricos ou quantidade de memória, simplifica a infraestrutura necessária para geração e compilação de código, pois apenas um único compilador precisa ser mantido e atualizado.

7.3.1 LaPSIman

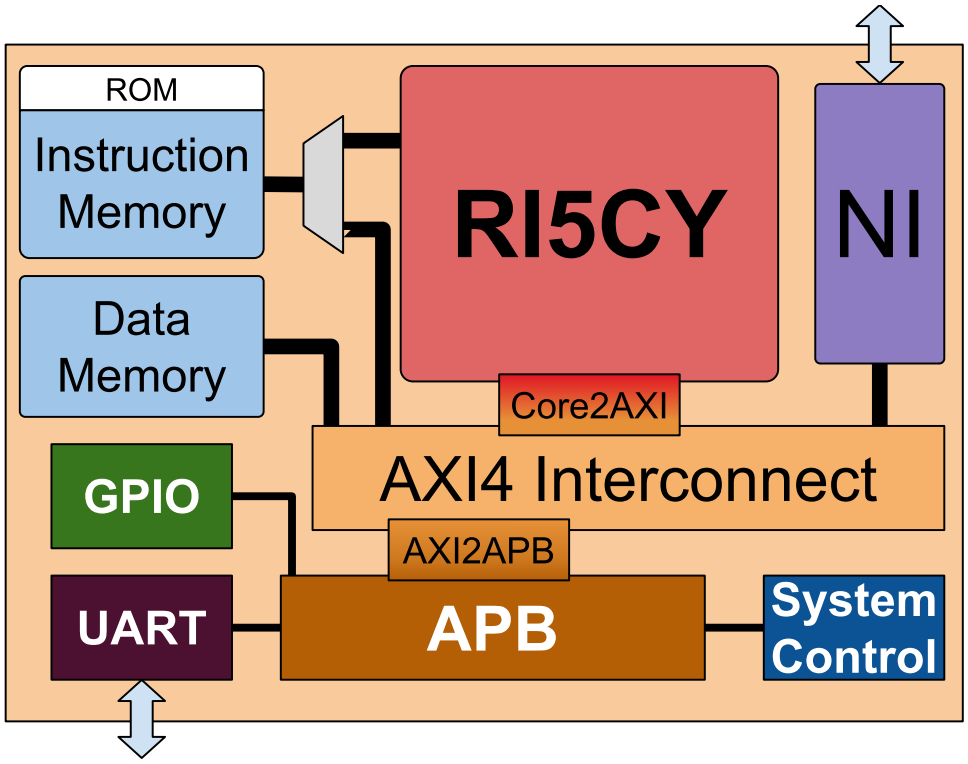
O propósito do LaPSIman é ser responsável pela conectividade externa e pelos canais de comunicação do sistema. Portanto, ele possui os periféricos necessários para realizar essas tarefas, como comunicação serial e GPIO. A aplicação de alto nível é implementada no LaPSIman, abstraindo a arquitetura do sistema para o mundo externo. Ele recebe os dados, prepara o comando a ser executado, divide os dados em lotes para serem processados, controla o fluxo de dados através dos *tiles* do MPSoC, reúne os resultados e os apresenta de volta ao solicitante.

Como visto na Figura 20, a arquitetura possui um *core* RI5CY conectado a um barramento AXI4 de alta velocidade onde são conectadas as memórias e os periféricos que exigem alto desempenho, como, por exemplo, a interface de rede. Esse barramento AMBA AXI4 possui 32-bits de largura de dados, e foi herdado do PULPino (TRABER *et al.*, 2016).

[1]Disponível em: https://github.com/pulp-platform/riscv

Ademais, ele possui os periféricos necessários para realizar o debug do sistema. Um barramento mais simples, do tipo APB está conectado ao AXI através de uma ponte construída exclusivamente para esse fim. Nesse barramento APB estão conectados periféricos de acesso lento, como Porta Serial, GPIO e os registradores de controle do sistema. Esses periféricos foram desenvolvidos para a plataforma PULPino e foram importados para o MPVue. Para facilitar o acesso a seus dados e controles, os periféricos são mapeados em memória, conforme mostrado na Figura 21.

Figura 20 – Arquitetura de Hardware do LaPSIman



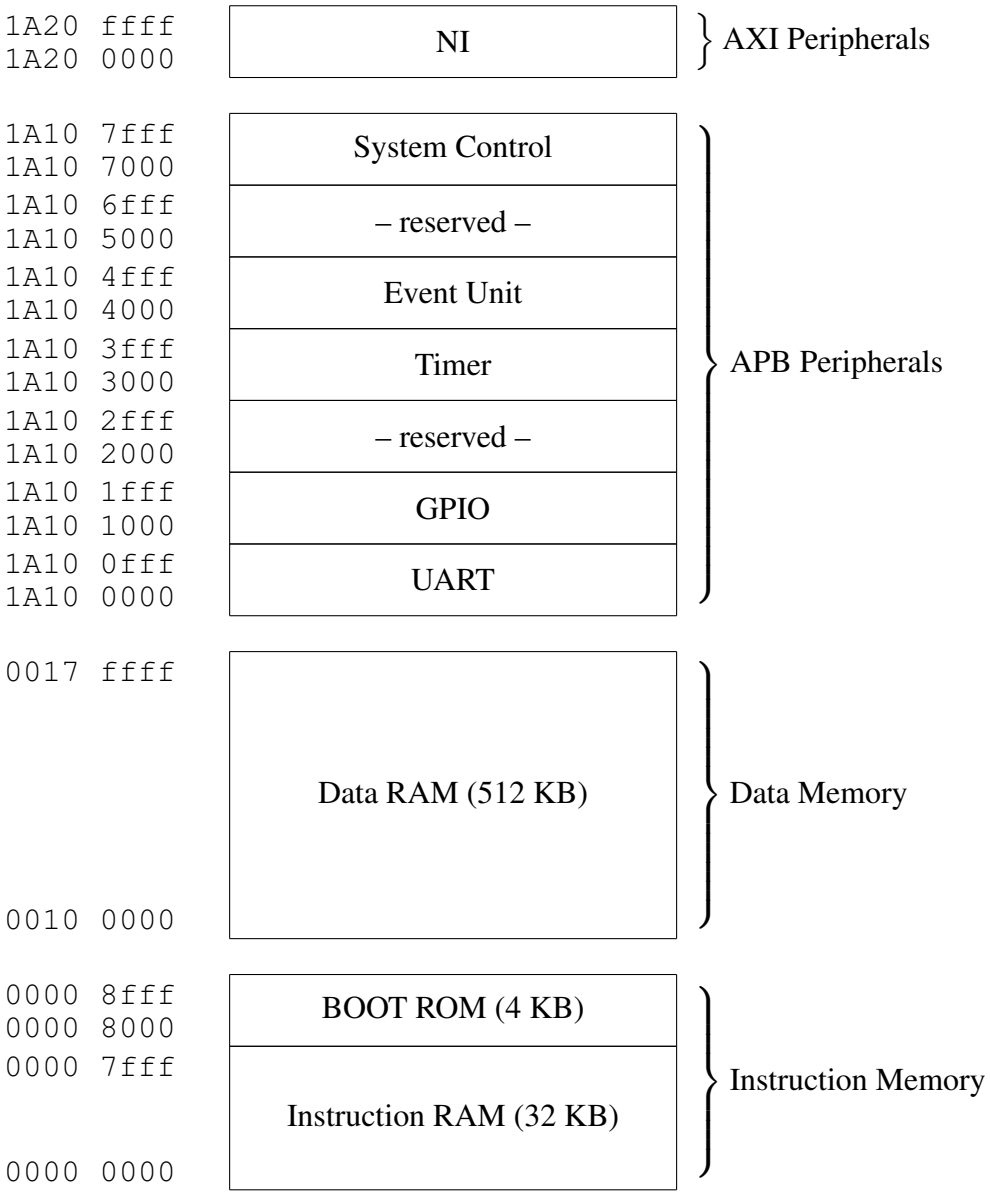
Fonte: do autor

Dessa forma, o debug do sistema é realizado através do LaPSIman, utilizando-se a porta serial para enviar comandos ao sistema e ler status, além da informação visual disponibilizada pelos LEDs ligados ao GPIO.

Como o processador RI5CY escolhido possui duas interfaces de memória – uma para *fetch* de instruções e outra para acesso à dados –, a memória foi dividida em duas partes, uma para dados e outra para instrução, conforme pode ser visto na Figura 20. A memória de instruções possui uma pequena ROM não volátil para inicialização do sistema. Nela um boot-loader pode ser armazenado para facilitar a carga de programas. A memória de instruções possui, também, uma parte de memória RAM para armazenamento de aplicações. Essa memória é mapeada tanto na interface de programa quanto na interface de dados. Isso permite uma agilidade para gravar aplicações durante o uso, permitindo uma rápida prototipação de software. A memória de programa é conectada através de um multiplexador à interface de programa do processador, evitando o uso concorrente do

barramento durante a execução do programa.

Figura 21 – Mapa de Memória do LaPSIman



Fonte: do autor

A memória de dados, por sua vez, é conectada ao barramento AXI4, visando a flexibilidade. Qualquer mestre do barramento pode acessá-la. Essa configuração permite flexibilidade de software e hardware em futuras versões, uma vez que novos componentes podem acessar diretamente a memória através do barramento, permitindo aplicações mais eficientes.

A interface de rede é conectada diretamente ao barramento AXI4, conforme mostrado na Figura 20. Dessa maneira, uma interface de alta velocidade é fornecida entre o processador e esse periférico. Quando um pacote precisa ser enviado, o processador configura os registradores mapeados em memória e armazena os dados na *fifo* de saída. Ao ser

ativada, a interface de rede envia o pacote. Uma interrupção, então, é ativada para informar o processador da conclusão do envio, e que um novo pacote pode ser enviado. Por outro lado, quando um pacote é recebido, uma interrupção é ativada informando do recebimento. Esse processo de interrupções permite ao processador que realize outras tarefas enquanto a interface lida com a comunicação.

Cada periférico possui uma entrada de interrupção fornecida pelo Controle do Sistema. Essa unidade fornece a opção de mascaramento das interrupções, bem como o gerenciamento de prioridade e o controle das interrupções.

A Figura 21 mostra o mapa de memória do LaPSIman. Nos endereços iniciais estão mapeadas as memórias de instrução, seguidas pela memória de dados. O barramento APB está ligado a partir do endereço 1A10_0000 do barramento AXI4, fazendo com que os periféricos mapeados em memória conectados no APB sejam remapeados para essa região de memória. Essa diferença entre os endereços da memória de dados e do barramento APB permite que até 26 MB de memória de dados possa ser mapeada linearmente, mesmo que nessa versão apenas 512KB sejam utilizados. Por fim, a interface de rede está mapeada com base no endereço 1A20_0000.

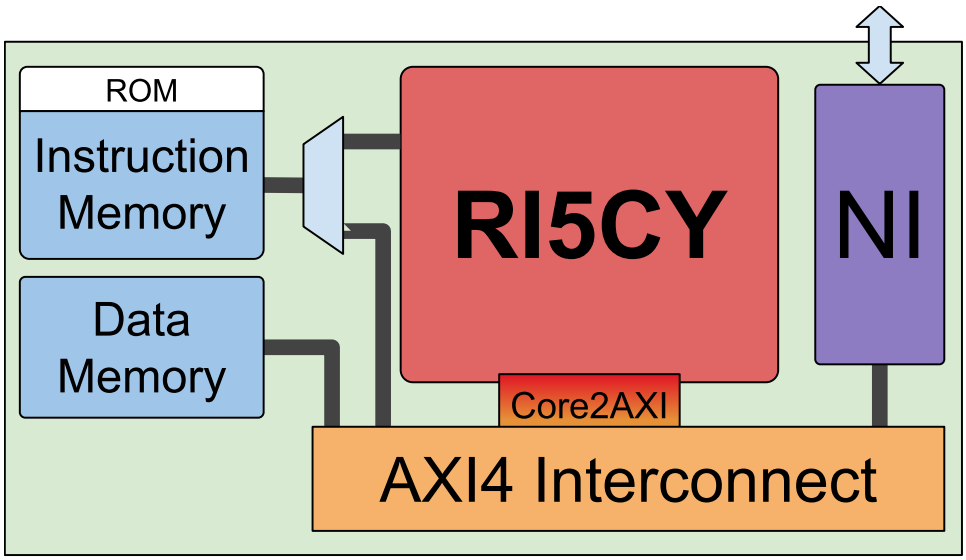
7.3.2 LaPSIno

O LaPSIno é um subconjunto minimalista do LaPSIman, sendo composto pelo processador, memória de programa e de dados, e uma Interface para a Rede em Chip, conforme mostrado na Figura 22. O processador é conectado à memória de programa através de um canal de leitura dedicado, enquanto a conexão à memória de dados e a escrita à memória de programa é realizada através de um barramento de interconexão AXI4. Uma ponte é utilizada para traduzir a transferência de dados para o protocolo da interface AXI4, já que o processador utiliza seu próprio protocolo. Um ou mais aceleradores podem ser acoplados ao barramento de dados para otimizar as funções em que o sistema será utilizado. Cada acelerador otimiza uma tarefa específica, ajudando a reduzir gargalos no fluxo de dados do programa. A integração dos aceleradores com o firmware da aplicação é fornecida através de drivers específicos.

Por ser um subconjunto do LaPSIman, as configurações de memória e barramento são herdados do mesmo. Uma memória ROM está presente para inicialização, bem como um canal exclusivo de acesso à memória de programa. A memória de dados é acessada através do barramento AXI4.

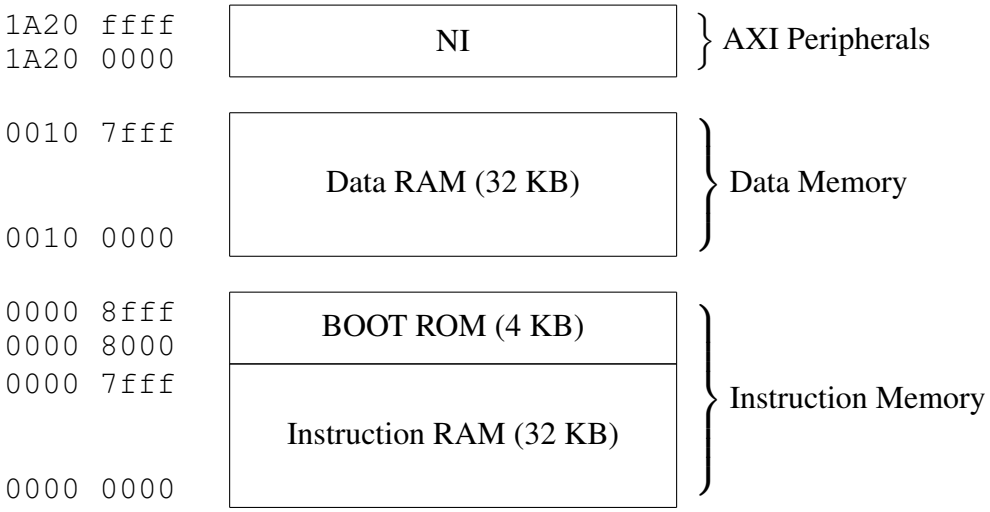
A Figura 23 mostra o mapa de memória do LaPSIno. Nos endereços iniciais estão mapeadas as memórias de instrução, seguidas pela de dados. Como não há barramento APB e para manter a compatibilidade de drivers com o LaPSIman, a interface de rede é mapeada com base no endereço 1A20_0000.

Figura 22 – Arquitetura de Hardware do LaPSIno



Fonte: do autor

Figura 23 – Mapa de Memória do LaPSIno



Fonte: do autor

7.4 Testes

Os testes do processador e da NoC foram realizados em seus respectivos projetos (TRABER; GAUTSCHI; SCHIAVONE, 2016; REINBRECHT *et al.*, 2016). Portanto, apenas testes sistêmicos no LaPSIman, LaPSInos e no MPVue foram necessários. Dessa maneira, foi desenvolvido um conjunto de simulações para verificar desde as funções básicas do sistema, como acesso à memória e busca de instruções, até funções mais complexas, como uma comunicação completa pela NoC utilizando interrupções no envio e na recepção. A Tabela 11 apresenta os testes realizados e o resultado obtido em cada um.

Tabela 11 – Testes realizados em simulação do sistema MPVue

Componente	Teste	Resultado
LaPSIman	Teste de Inicialização	Passou
	Teste de Timer	Passou
	Teste de Timer com Interrupção	Passou
	Teste de Comunicação Serial	Passou
	Teste de Comunicação NoC	Passou
	Teste de Gravação de Programa pela Serial	Passou
LaPSIno	Teste de Inicialização	Passou
	Teste de Comunicação NoC	Passou
	Teste de Comunicação NoC com Interrupção	Passou
	Teste de Gravação de Programa pela NoC	Passou
MPVue	Teste de Inicialização	Passou
	Teste de Comunicação pela NoC	Passou

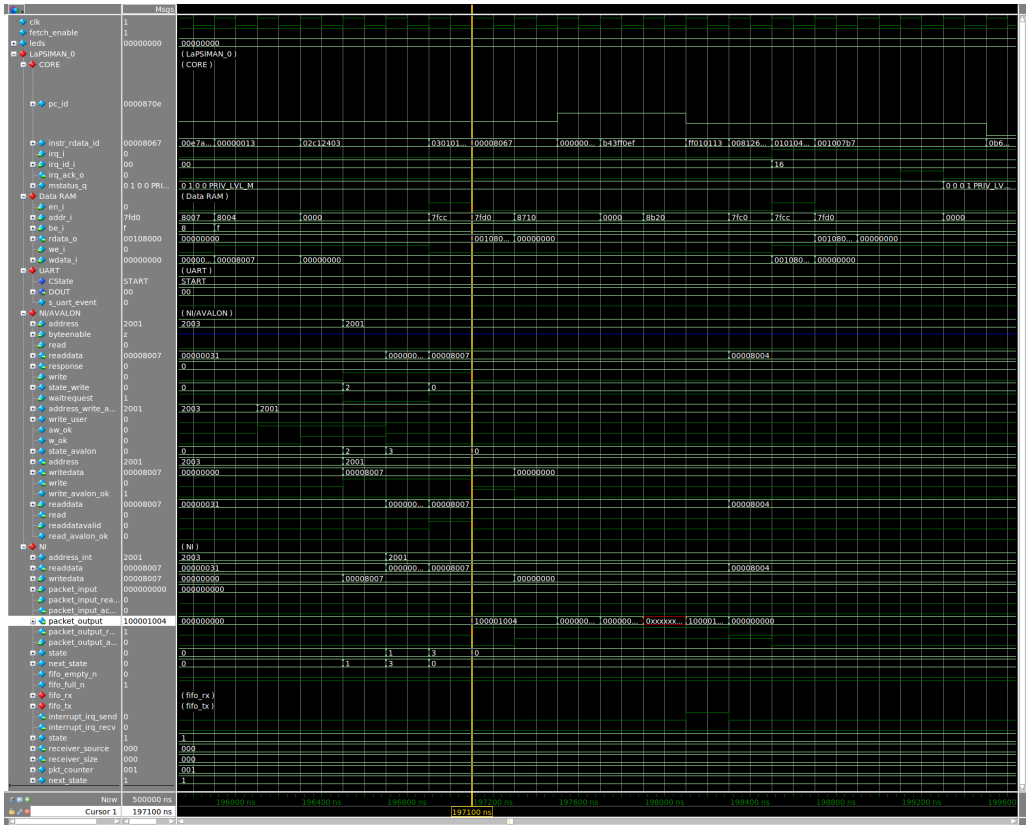
A Figura 24 mostra o resultado da simulação do envio de um pacote para a NoC pelo LaPSIman. É possível ver os sinais de acesso à interface de comunicação, o início da transmissão em *packet_output*. Ao terminar a transmissão, o sinal *irq_i* é ativado em conjunto com o *irq_id*, indicando a presença de uma interrupção. Em seguida, o tratamento dessa interrupção é indicado pelo salto para o endereço de tratamento da interrupção de conclusão de envio da NoC.

A Figura 25 mostra o resultado da simulação do recebimento de um pacote vindo da NoC para o LaPSIman. É possível ver o pacote chegando através da interface *packet_input*. Ao terminar o recebimento, o sinal *irq_i* é ativado, em conjunto com o *irq_id*, indicando a presença de uma interrupção. Na sequência, o tratamento dessa interrupção é indicado pelo salto para o endereço de tratamento de interrupção de recepção da NoC.

7.5 Conclusão do Capítulo

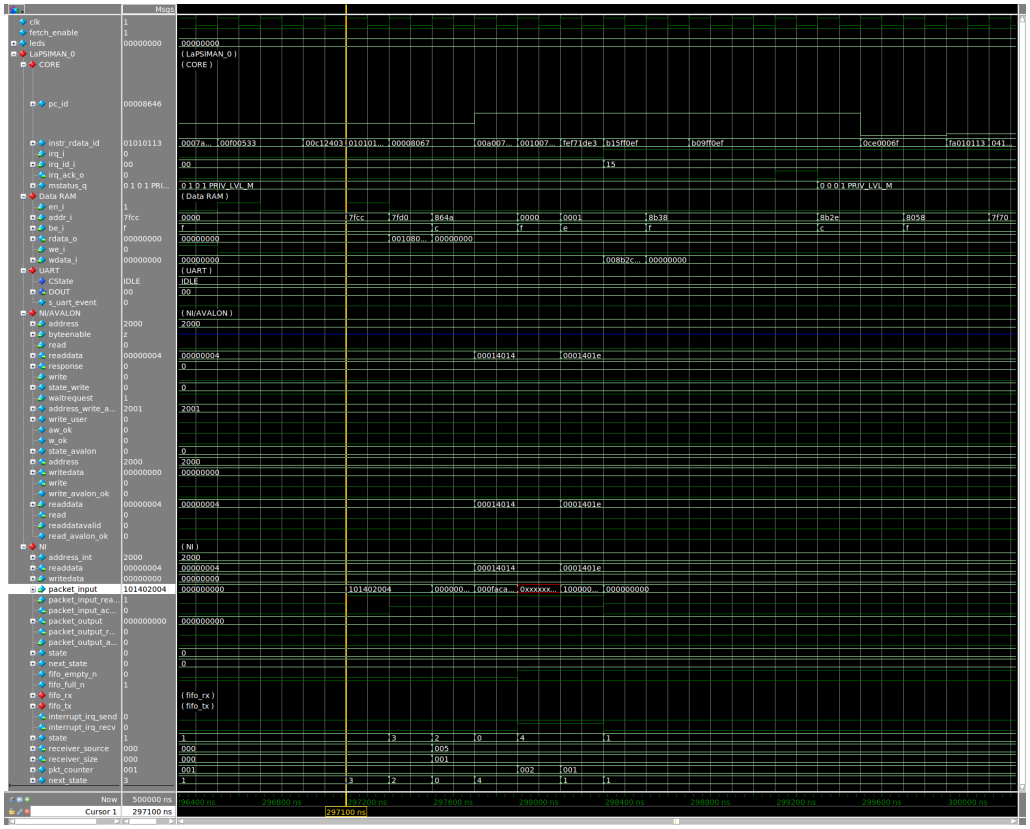
A arquitetura do MPVue foi apresentada, detalhando as funcionalidades da rede em chip e dos *tiles*. Os testes sistêmicos comprovam o funcionamento completo da solução e habilita o uso do mesmo em projetos mais complexos.

Figura 24 – Simulação do LaPSIman - Envio de dados pela NoC



Fonte: do autor

Figura 25 – Simulação do LaPSIman - Recebimento de dados pela NoC



Fonte: do autor

8 ARQUITETURA DE SOFTWARE EMBARCADO

Para a implementação de software para MPSoCs, normalmente o código é compilado exclusivamente para a aplicação rodando na plataforma específica (SIEVERS *et al.*, 2015; DAVIDSON *et al.*, 2018; DOGAN *et al.*, 2012). Dessa forma, cada plataforma é construída baseada em características específicas do hardware e a implementação é dependente de um conjunto de ferramentas (compilador, linker e montador) criados para a plataforma. Isso traz dificuldade quando se deseja portar a aplicação de uma plataforma para outra, pois, mesmo que seja utilizada a mesma plataforma, a simples mudança de parâmetros, como número de *tiles* ou seus reposicionamentos, exige uma recompilação completa.

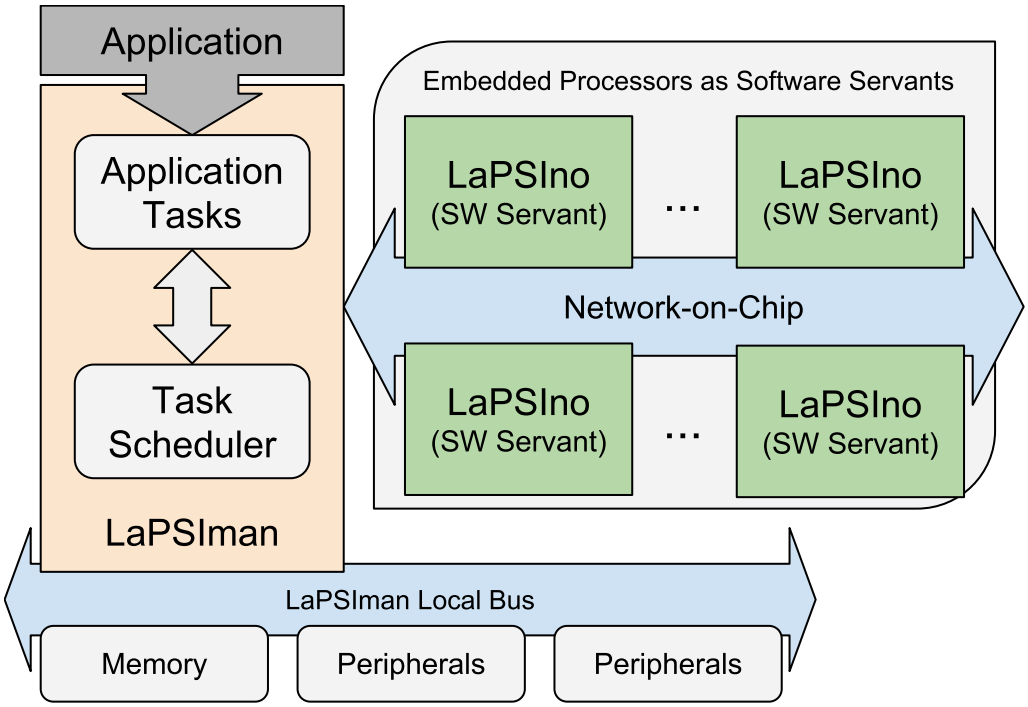
Por outro lado, Wang propõe uma arquitetura orientada a serviços (SOA) para MPSoCs, que é chamada de SoSoC (WANG *et al.*, 2017). Nessa proposta, cada elemento processador, ou *servant*– que pode ser um processador, um *tile*, um acelerador em hardware, ou outros – possui uma API que permite disponibilizar serviços pré-definidos que podem ser utilizados como blocos construtores de qualquer aplicação paralela. Por fim, é mostrado que a complexidade de integração pode ser reduzida quando as interfaces estão bem definidas.

8.1 Programação Orientada a Serviços

A metodologia utilizada para desenvolver programas para um sistema com arquitetura orientada a serviços é chamada programação orientada a serviços (SOP) (PAPAZOGLOU *et al.*, 2007). Assim, inspirado em Wang, o MPVue implementa aplicações de visão computacional em um modelo de programação orientada a serviços. Essas aplicações consistem em alguns serviços, também chamados de tarefas, que executam em grandes dados (imagens ou vídeos) com um grande nível de independência entre eles. Cada *servant* oferece um repertório de serviços que podem ser chamados através da API pré-definida.

Como mostrado na Figura 26, a aplicação é formada por um conjunto de tarefas que são gerenciadas pelo LaPSIman, construído para ser o controlador do sistema. Ele conhece o estado de todos os elementos do sistema e consigna as tarefas sequencialmente

Figura 26 – Arquitetura Orientada a Serviços



Fonte: do autor

de forma otimizada, usando um modelo de agendador de tarefas. Se as tarefas forem independentes entre si, então a execução pode ser feita fora de ordem naturalmente. Nesse modelo, cada LaPSIno é um *servant*, oferecendo seus serviços ao LaPSIman.

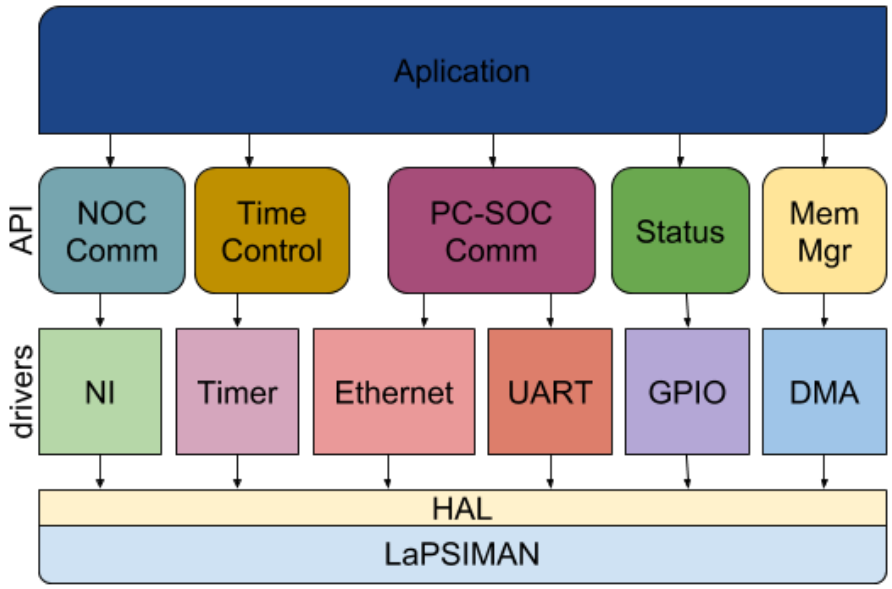
8.2 Arquitetura de Software do MPvue

A arquitetura de software do MPVue é focada em alta performance e re-usabilidade. Os módulos de software, chamados de drivers, são integrados ao hardware de forma transparente e podem ser inseridos ou removidos do projeto em tempo de compilação, dependendo exclusivamente do uso que se quer dar à plataforma.

O software foi concebido em um modelo de desenvolvimento *bottom-up*, utilizando um esquema de interrupções e *callbacks* que permite que o código prioritário seja processado com latência baixa sem sacrificar o contexto de uma parte complexa do código. Isso busca reduzir a necessidade de *pooling* e tempo gasto em espera de resposta de periféricos ou outros elementos do sistema. A hierarquia de software é construída de forma que as funções mais simples e, também, mais executadas, são otimizadas especialmente para o hardware. Tais funções, então, são disponibilizadas em módulos abaixo de uma camada de abstração. O objetivo dessa camada é ampliar a programabilidade da plataforma, uma vez que abstém o desenvolvedor da aplicação de detalhes de hardware que não são úteis para sua implementação. Dessa maneira, o desenvolvedor pode extrair o máximo desempenho do sistema. Módulos foram desenvolvidos para a interface de rede, para os timers,

para ethernet, comunicação serial e GPIO. A Figura 27 mostra um exemplo de integração de software por uma aplicação, que utiliza os módulos disponibilizados.

Figura 27 – Arquitetura de software baseada em módulos



Fonte: do autor

8.3 Fluxo de desenvolvimento

No modelo de programação orientada a serviços, não há necessidade de um compilador único e complexo para o sistema inteiro. Cada programa desenvolvido para um LaPSIno é um conjunto de serviços disponibilizados e que são desenvolvidos independentemente, usando a API pré-definida. Todos os LaPSInos podem compartilhar, ou não, o mesmo programa, dependendo dos serviços que se deseja disponibilizar. Com esse método, cada LaPSIno possui seu próprio conjunto de serviços que pode ser otimizado independente dos outros. O LaPSIman pode escolher qual tarefa é executada em cada LaPSIno baseado no tempo de execução, além de critérios individualizados de performance. Portanto, o programa do LaPSIman implementa a aplicação desejada, usando a API para otimizar o desempenho geral da solução.

Uma vez que o LaPSIman e os LaPSInos compartilham a mesma arquitetura de processador, o software para ambos pode ser compilado por qualquer compilador compatível com o conjunto de instruções do RISC-V, opcionalmente configurado para utilizar as instruções contidas nas extensões: multiplicação e divisão (M) e instruções compactas (C). Se as instruções específicas do RI5CY forem desejadas, é disponibilizado um compilador GCC específico para ele, tornando o uso das funções aceleradas transparente ao programador.

8.4 Modelo de Comunicação

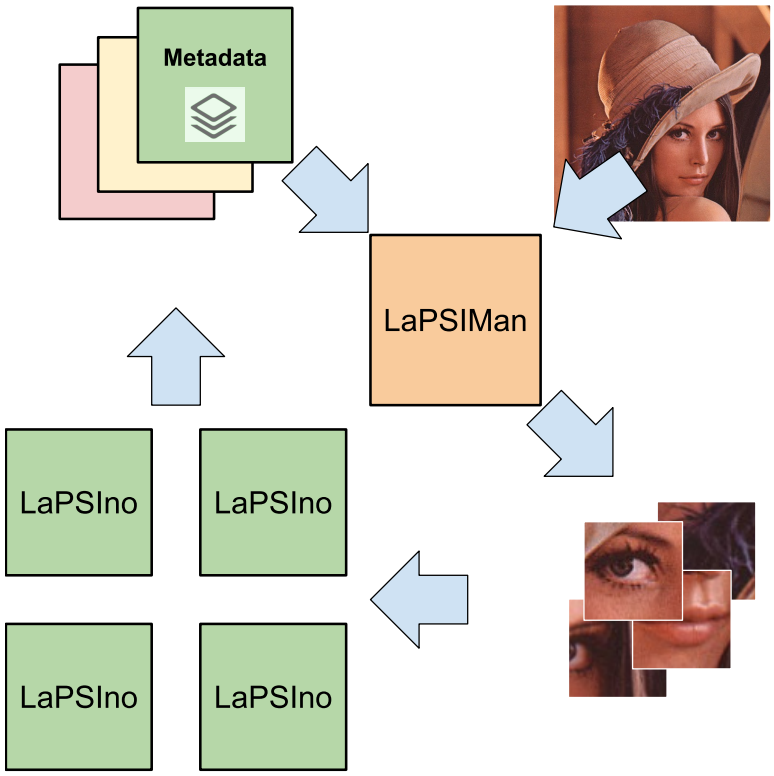
Uma das formas de implementar a Programação Orientada a Serviços é baseada em disponibilizar os dados ao *servant* e solicitar que um serviço seja executado sobre esse dado. Uma vez que o LaPSIman não possui acesso direto à memória privada dos LaPSInos, algumas partes dessa memória são reservadas e chamadas de contextos. Cada contexto possui um tamanho fixo, e pode ser lido ou escrito por comandos desenvolvidos para esse fim. No MPVue, os dados são disponibilizados através desses contextos, cujo conteúdo pode ser alterado através de comandos específicos para esse fim. Assim, o LaPSIman pode armazenar os dados a serem processados de forma indireta na memória do LaPSIno. Portanto, um modelo de comunicação eficiente é necessário para transportar essa grande quantidade de dados, minimizando a chance desse transporte se tornar o gargalo do sistema. Esse modelo permite, ainda, que, se algum conjunto de dados já foi anteriormente enviado para um LaPSIno durante uma computação, não seja necessário transferir esse conjunto de dados novamente. O número e o tamanho dos contextos pode ser determinado durante a compilação – 5 contextos de 4KB cada foram utilizados nesse trabalho.

O fluxo normal de trabalho em visão computacional corresponde ao processamento encadeado de uma sequência de imagens, que são recebidas por um canal de comunicação e informações são geradas e armazenadas ou transmitidas a outros dispositivos. No MPVue, conforme mostrado na Figura 28, as imagens são recebidas pela interface Ethernet e armazenadas diretamente na memória. Uma nova rodada então é iniciada: a imagem é dividida em blocos de tamanho fixo e enviada a cada processador para ser processada. Cada LaPSIno recebe sua parte a processar através da NoC. Após terminar o seu processamento, os dados gerados são enviados à próxima etapa e o processador fica pronto para uma nova operação, podendo receber um novo bloco ou processar os dados gerados por outros processadores.

O nível de paralelismo pode ser definido pelo número de núcleos e por sua independência para tomar decisões sobre o que e como fazer o processamento, inclusive interagindo localmente – com processadores próximos – para eliminar gargalos no gerente do sistema.

Um modelo de comunicação bidirecional utilizando comando resposta foi implementado. Assim, qualquer comunicação entre diferentes *tiles* é permitida com confirmação de recebimento. Sempre que uma tarefa precisa ser realizada, o LaPSIman envia um comando contendo o endereço do LaPSIno que deve executá-la, com um código identificando o serviço e ponteiros para os contextos a serem utilizados.

Figura 28 – Fluxo de dados do MPVue



Fonte: do autor

8.5 Conclusão do Capítulo

A arquitetura de software para o MPVue foi apresentada. A arquitetura orientada a serviços foi introduzida e seu funcionamento dentro do sistema foi explicado, bem com o fluxo de desenvolvimento. Por fim, o modelo de comunicação entre *tiles* mostrou de forma geral a ideia da divisão das tarefas e a forma que essas tarefas são delegadas.

9 MPVUE - AVALIAÇÃO DE DESEMPENHO

De forma a verificar a flexibilidade da plataforma, além de apresentar alternativas para diferentes necessidades, duas configurações são propostas para comparação: *MPVue 3x3* e *MPVue 4x4*. O *MPVue 3x3* é menor, porém poderoso, contendo 9 elementos processadores com baixa latência de comunicação. Alternativamente, o *MPVue 4x4* contém 16 elementos processadores e utiliza ao máximo os recursos da placa KC705 para entregar o máximo desempenho.

9.1 Implementação

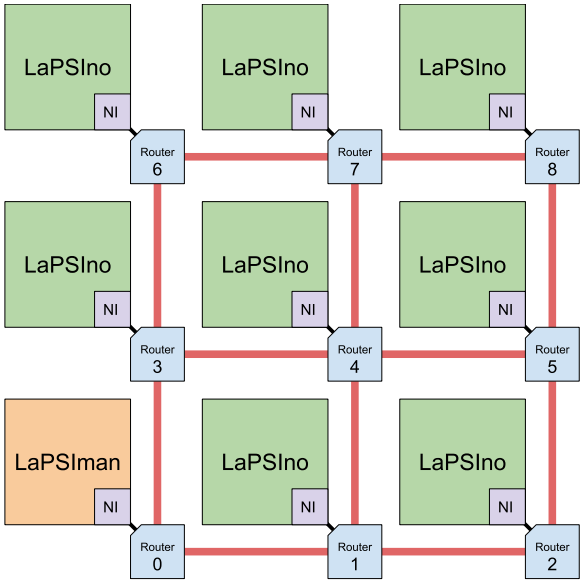
A plataforma MPVue permite a possibilidade de alterar a estrutura de hardware antes da síntese, de forma a maximizar a eficiência da aplicação final. Isso é obtido utilizando scripts customizados de síntese. Assim, qualquer desenvolvedor pode escolher os parâmetros de hardware, como o tamanho da NoC, o número de processadores embarcados, tamanho e tipo de memória, bem como os periféricos a serem utilizados na aplicação desejada. A plataforma é então autonomamente gerada utilizando esses parâmetros.

9.2 *MPVue 3x3*

A configuração *MPVue 3x3* é composta por uma NoC 3x3, contendo um LaPSIman e LaPSInos, como mostrado na Figura 29. O LaPSIman é localizado no endereço 0 da NoC. Sua memória de instrução foi configurada com 32KB e sua memória de dados com 512KB. Os LaPSInos foram localizados nos demais endereços, cada um configurado com 32KB de memória de instrução e 32KB de memória de dados.

A Tabela 12 mostra a utilização dos recursos da placa pela configuração *MPVue 3x3*, separadas pelos diferentes tipos de estrutura da FPGA. Ela mostra que um LaPSIno é menor que o LaPSIman em 42% quando se compara quantidade de flipflops e 23% em utilização de LUTs. Essa diferença se deve ao uso do barramento APB e dos periféricos que estão disponíveis no LaPSIman. Além disso, a utilização de BRAM é dependente da memória de instrução e de dados configuradas para cada processador, e isso é refletido na tabela.

Figura 29 – Configuração *MPVue 3x3*



Fonte: do autor

Tabela 12 – Utilização total de recursos da placa KC705 pela configuração *MPVue 3x3*

	FF's	LUT	BRAM	DSPs	BUFG
LaPSIman	7736	13909	136	6	0
LaPSIno	4413	10613	16	6	0
NoC 3x3	2014	7969	0	0	0
Total	45054	106756	264	54	2
% utilização	11,82%	52,38%	59,33%	6,43%	6,25%

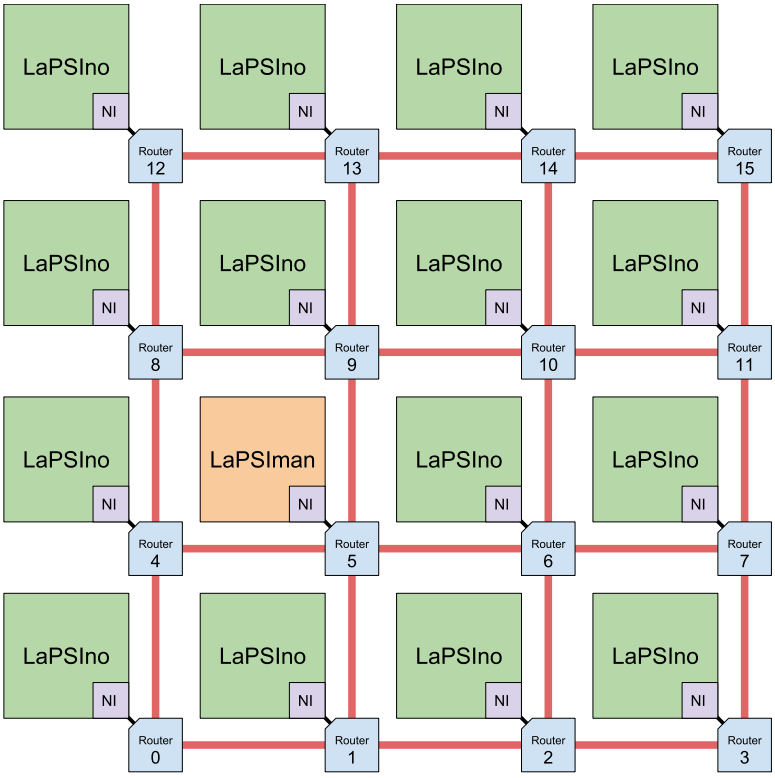
Na configuração *MPVue 3x3*, uma grande quantidade de espaço livre está disponível na placa para expandir o sistema. Mais processadores podem ser adicionados, como pode ser visto no *MPVue 4x4*, ou, ao contrário, mais memória pode ser adicionada a cada LaPSIno, ou periféricos podem ser adicionados aos processadores.

9.3 *MPVue 4x4*

A configuração *MPVue 4x4* é composta por uma NoC 4x4, contendo 1 LaPSIman e 15 LaPSInos, como mostrado na Figura 30. O LaPSIman é localizado no endereço 5 da NoC, de forma a minimizar a máxima latência de comunicação entre ele e os outros LaPSInos. Sua memória de instrução é configurada com 32KB e a memória de dados com 512KB. Os LaPSInos foram localizados nos demais endereços, cada um configurado com 32KB de memória de instrução e 32KB de memória de dados.

A Tabela 13 mostra a utilização dos recursos da placa pela configuração *MPVue 4x4*, separadas pelos diferentes tipos de estrutura do FPGA. O tamanho dos processadores é

Figura 30 – Configuração *MPVue 4x4*



Fonte: do autor

o mesmo do que na configuração *MPVue 3x3*, então sua relação permanece a mesma. A utilização de recursos pela NoC aumenta na configuração *MPVue 4x4*, quando comparada com a *MPVue 3x3*, já que mais *roteadores* e suas interconexões são necessários. Além disso, o fato de haver mais LaPSInos na configuração *MPVue 4x4* faz com que a quantidade total de recursos utilizada aumente na mesma proporção.

Tabela 13 – Utilização total de recursos da placa KC705 pela configuração *MPVue 4x4*

	FF's	LUT	BRAM	DSPs	BUFG
LaPSIman	7736	13827	136	6	0
LaPSIno	4413	10572	16	6	0
NoC 4x4	3743	15162	0	0	0
Total	77674	187349	376	96	2
% utilização	19,06%	91,93%	84,49%	11,43%	6,25%

Apesar da Tabela 13 mostrar uma alta utilização de memória BRAM, muito dessa utilização se deve aos 512KB de memória de dados alocada ao LaPSIman. Isso pode ser ajustado ao tempo da síntese, se necessário, como, por exemplo, em aplicações onde os LaPSInos possam necessitar mais memória para uma tarefa específica.

9.4 Comparativo de Aceleração

Para estabelecer uma base de comparação entre diferentes trabalhos, um conjunto de aplicações foram selecionadas para avaliação. Os algoritmos de transformada rápida de Fourier (*FFT*) e filtro passa-baixas (*lowPass*) foram escolhidos pois são comuns entre diferentes soluções computacionais de processamento de imagens. Esses algoritmos foram implementados baseados na plataforma StreamIt (THIES; KARCZMAREK; AMARASINGHE, 2002), que propõe um conjunto de aplicações de processamento de sinais com potencial de serem executadas em paralelo. Essa escolha se deveu ao fato de (SIEVERS *et al.*, 2015; AX *et al.*, 2018) utilizarem a plataforma ara realizar suas medidas de desempenho. Assim, a comparação do MPVue com essas plataformas pode ser feita de forma direta. O compilador padrão GCC para RISC-V disponibilizado por (IONESCU, 2017) foi usado para compilar os códigos tanto para o LaPSIman quanto para os LaPSInos. A otimização do GCC foi estabelecida para foco em desempenho (-O2).

9.4.1 FFT

Para o cálculo da FFT, 32KB de dados foram transformado usando FFT 1-D, tendo o desempenho comparado em diferentes tamanhos de vetores: 64, 128, 256 e 512 pontos. Em cada rodada de cálculo, um grupo de vetores, chamados lote, é enviado a um LaPSIno e transformado. Assim, cada lote pode ser calculado em paralelo com os outros, enviados a outros LaPSInos.

A comparação é feita utilizando lotes contendo de 1 a 8 vetores. O tamanho máximo do lote é limitado ao tamanho do contexto, não podendo extrapolar a região de memória estipulada para ele. Além disso, existe um limite de quantidade de lotes que o LaPSIman consegue gerenciar, o que estabelece um limite mínimo ao lote. A Tabela 14 mostra o número máximo de lotes e o tamanho máximo do contexto para cada tamanho de lote, em cada tamanho de vetor. Como o LaPSIman está configurado para gerenciar até 128 lotes, as quantidades de lotes marcadas em vermelho mostram as opções que ultrapassam esse valor. De mesmo modo, como o tamanho de um contexto no LaPSIno é de 4096 bytes, os tamanhos de contextos máximos marcados em vermelho mostram as opções que ultrapassam esse limite. As opções marcadas em cinza são aquelas que não são utilizadas nos testes, em razão dos limites acima expostos.

Antes do cálculo, o LaPSIman converte os dados para ponto flutuante e os armazena como a parte real do vetor, sendo a parte imaginária zerada. Todos os cálculos são feitos em ponto flutuante, retornando o resultado em dois vetores: um contendo a parte real e o outro a imaginária. A comunicação externa com o sistema, bem como a parte inicial de conversão para ponto flutuante não foi considerada no cálculo de desempenho.

Conforme visto na Seção 4.3, o cálculo de aceleração é o comparativo entre o desempenho do sistema inteiro e o desempenho de uma única CPU rodando a aplicação. O

Tabela 14 – Número de lotes e Tamanho de contexto para cada tamanho de lote em cada tamanho de vetor

Tamanho do Lote	512		256		125		64	
	Núm lotes	Tamanho contexto	Núm lotes	Tamanho contexto	Núm lotes	Tamanho contexto	Núm lotes	Tamanho contexto
1	64	2048 B	128	1024 B	256	512 B	512	256 B
2	32	4096 B	64	2048 B	128	1024 B	256	512 B
3	22	6144 B	43	3072 B	86	1536 B	171	768 B
4	16	8192 B	32	4096 B	64	2048 B	128	1024 B
5	13	10240 B	26	5120 B	52	2560 B	103	1280 B
6	11	12288 B	22	6144 B	43	3072 B	86	1536 B
7	10	14336 B	19	7168 B	37	3584 B	74	1792 B
8	8	16384 B	16	8192 B	32	4096 B	64	2048 B

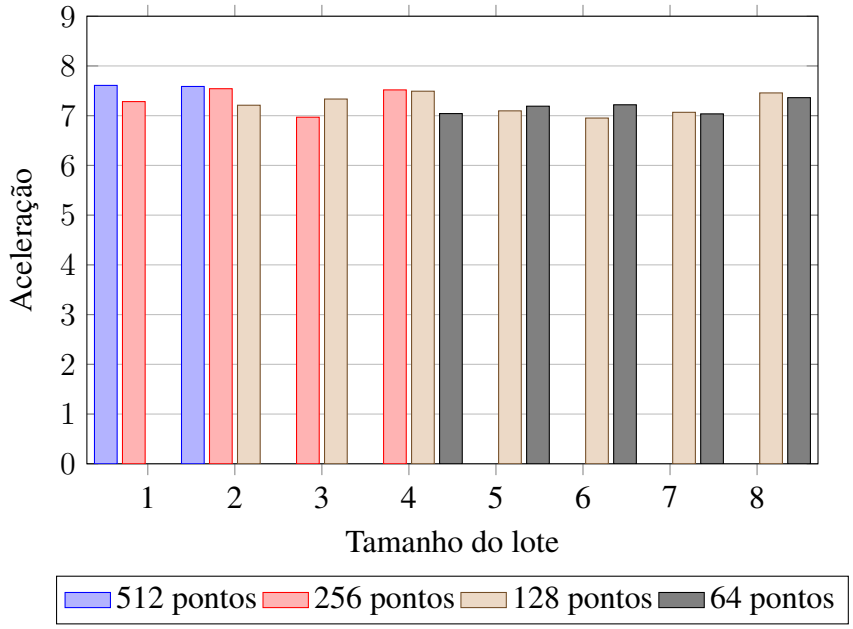
desempenho da CPU única foi medido rodando a aplicação somente no LaPSIman. Os testes de desempenho foram rodados em ambos as configurações *MPVue 3x3* e *MPVue 4x4*.

A Figura 31 mostra o resultado de aceleração na configuração *MPVue 3x3*. Ela apresenta a aceleração para diferentes tamanho de vetores – 512, 256, 128 e 64 pontos – para cada tamanho de lote. Os resultados mostram que para a FFT de 64 pontos – realizada utilizando os lotes de tamanho 4 a 8 – a aceleração está entre 7,03x e 7,36x. Para a FFT de 128 pontos – para lotes de 2 a 8 – a aceleração também oscila próximo de 7x, mostrando resultados entre 6,95x e 7,49x. Devido ao tamanho do contexto, a FFT de 256 pontos foi limitada a lotes de tamanho igual a 4 vetores. A aceleração nesse caso atingiu 7,54x nesse lote de 4 vetores. Por fim, a FFT de 512 pontos, executada somente em lotes de 1 e 2 vetores, traz os melhores resultados, 7,6x e 7,58x, respectivamente. Nesse tamanho de vetor o sistema tem o maior ganho no paralelismo, uma vez que mais dados são processados em paralelo, reduzindo as perdas de comunicação, como é melhor discutido na Seção 9.4.

Já a Figura 32 mostra o resultado de aceleração na configuração *MPVue 4x4*. Os tamanhos de vetores são os mesmos do teste anterior, assim como os tamanhos de lotes. Nessa configuração é possível notar uma diferença maior de aceleração nos diferentes tamanhos de lotes. Os resultados mostram que quando há uma distribuição assimétrica de lotes entre os LaPSInos o desempenho pode ser reduzido em até 20%. Tal fenômeno tem como causa o fato de que quando não há lotes suficientes para todos os LaPSInos, alguns ficam ociosos, esperando os outros terminarem seu processamento. Apesar disso, a aceleração ainda está acima de 10x em todos os casos, exceto um, atingindo aceleração acima de 12x para todos os tamanhos de vetor em pelo menos uma configuração de tamanho de lote .

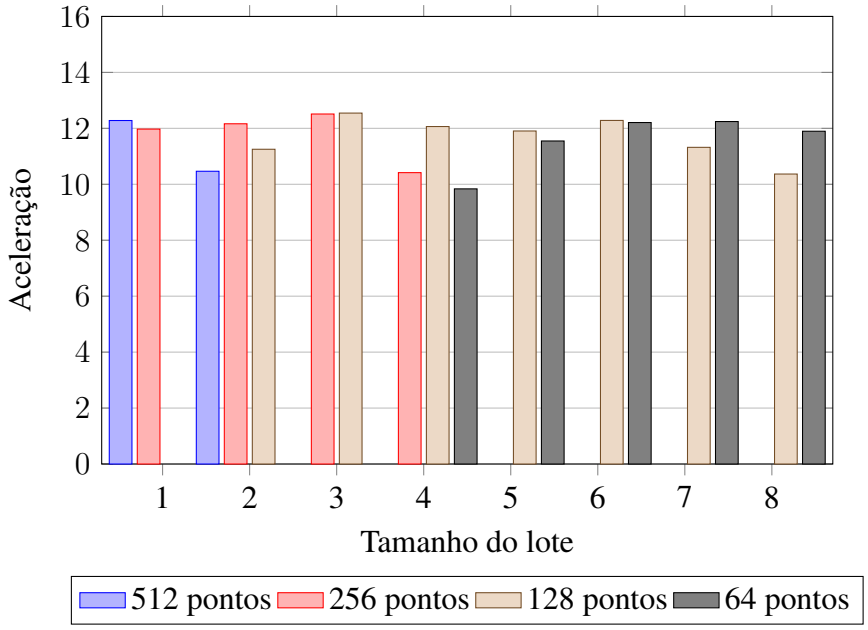
Esses resultados foram comparados com soluções recentes. A Tabela 15 mostra a

Figura 31 – Aceleração do cálculo da FFT no *MPVue 3x3*



Fonte: do autor

Figura 32 – Aceleração do cálculo da FFT no *MPVue 4x4*



Fonte: do autor

comparação entre o desempenho do MPVue e as soluções propostas por Airoldi (AIROLDI; GARZIA; NURMI, 2010) e Garibotti (GARIBOTTI *et al.*, 2013), que utilizam o mesmo tamanho de rede em chip que a configuração *MPVue 3x3*.

Tabela 15 – Comparativo de aceleração do algoritmo de FFT entre as soluções

	NoC	Núm cores	Aceleração (por número de pontos da FFT) 64	128	256	512	2048	N/D
Garibotti	3x3	9						4,6x
Airoldi	3x3	9	5,89x				6,89x	
MPVue	3x3	9	7,36x	7,49x	7,54x	7,60x		
	4x4	16	12,24x	12,54x	12,51x	12,28x		
Bahn	4x4	16	2,64x	4,84x	7,82x	9,73x	12,73x	
Sievers	2x2	16	6,9x					
	2x1	16	7,9x					
Ax	4x2	32	23,1x					
	2x2	32	22,6x					
	2x1	32	23,9x					

Apesar de Garibotti (GARIBOTTI *et al.*, 2013) apresentar sua solução utilizando uma rede em chip 3x3, ele utiliza apenas 8 processadores para realizar a transformada, enquanto um cuida apenas de gerenciar o sistema. Diferente do MPVue, ele divide as tarefas em *threads*, e cada uma dessas *threads* é responsável por uma parte do cálculo, dentro do mesmo vetor. Não há informação de quantidade de pontos utilizados no vetor de cálculo da FFT, apenas a informação do tamanho de cache para armazenar os dados (16KB). Seu melhor resultado traz uma aceleração de 4,6x, mesmo assim, é inferior ao pior resultado do MPVue com 7,36x na configuração de 64 pontos.

Airoldi (AIROLDI; GARZIA; NURMI, 2010) apresenta resultados para cálculo de FFT em vetores de 64 e 2048 pontos, em uma rede em chip 3x3. Apesar de ser um sistema desenvolvido exclusivamente para cálculo de FFT, sua aceleração para um vetor de 64 pontos de 5,89x é inferior ao resultado do MPVue de 7,36x para a mesma configuração. Mesmo no vetor maior, onde seu desempenho aumenta, a solução de Airoldi atinge aceleração de 6,89x, ainda abaixo da aceleração obtida no MPVue.

Bahn (BAHN; YANG; BAGHERZADEH, 2008) possui resultados para tamanhos de vetor iniciando em 32 até 32768 pontos, o que é infactível para o MPVue, conforme mostrado na Tabela 14. Entretanto, nos tamanhos de vetores entre 64 e 512 pontos é possível estabelecer comparação direta. Apesar de ter desenvolvido um algoritmo para execução paralela da FFT, ele se beneficia apenas de vetores com grande quantidade de pontos, atingindo aceleração maior que a máxima do MPVue somente em vetores de 2048 pontos. Seu melhor resultado é atingido na FFT de 16384 pontos, em que sua aceleração atinge 14,75. No seu melhor resultado comparável com o MPVue, na FFT de 512 pontos, sua aceleração na configuração de rede em chip 4x4 é de apenas 9,73x, o que se compara

com 12,28x do MPVue.

Sievers (SIEVERS *et al.*, 2015) apresenta os resultados de sua plataforma baseada em CoreVA. A aceleração é avaliada pelo *benchmark* StreamIt, que utiliza o cálculo de FFT de 64 pontos por padrão. Sievers traz duas configurações de rede em chip– 2x2 e 2x1 – usando 16 elementos processadores distribuídos entre os *tiles*. Apesar de não possuir uma configuração igual à do MPVue, o número de elementos processadores permite uma comparação direta com a configuração *MPVue 4x4*. O melhor resultado de Sievers, 7,9x na NoC 2x1, pode ser comparado com o resultado de 12,24x do MPVue no mesmo tamanho de vetor. Ax (AX *et al.*, 2018) estendeu o trabalho, aumentando o número de elementos processadores de 16 para 32, distribuindo-os entre os *tiles* em configurações de rede em chip 4x2, 2x2 e 2x1. Seu melhor resultado, na configuração 2x1, possui aceleração de 23,9x, o que não pode ser diretamente comparado à aceleração obtida pelo MPVue devido à diferença no número de elementos processadores nas duas soluções.

Conforme visto na Seção 4.3, para se poder comparar diferentes implementações que não utilizam o mesmo número de elementos processadores, Airoldi (AIROLDI; GARZIA; NURMI, 2010) utiliza o cálculo de eficiência paralela. Ela é obtida dividindo-se a aceleração pelo número de elementos processadores. A Figura 33 mostra a eficiência das soluções contidas na Tabela 15. Sievers possui eficiência próxima de 50% e Ax atinge no máximo 74,9%. Airoldi e Bahn mostram um aumento de eficiência, proporcional ao tamanho do vetor ao qual se calcula a FFT, atingindo, no vetor de 2048 pontos, eficiência de 76,6% e 79,6%, respectivamente.

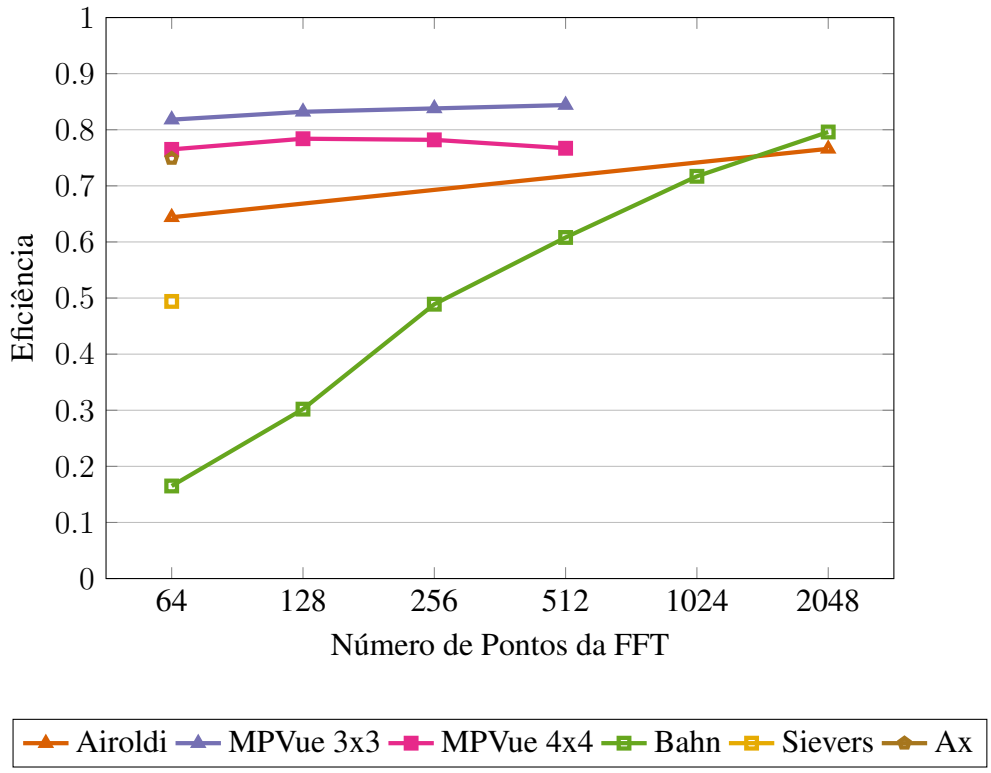
O MPVue, por outro lado, apresenta um resultado mais constante, com pouca influência do tamanho do vetor. *MPVue 3x3* possui eficiência acima de 80% em qualquer tamanho de vetor, chegando a 84,4% no vetor de 512 pontos. *MPVue 4x4* possui uma eficiência um pouco menor, com resultados entre 75% e 80%. No cálculo de vetor de 64 pontos, seu resultado de 76,5% é muito próximo do melhor resultado de Ax, ultrapassando-o em 1,6 pontos percentuais. Nos tamanhos de vetores em que foi calculado, possui um desempenho melhor que Bahn, com resultado de 78,2% no vetor de 512 pontos contra 60,8% de Bahn. O melhor resultado do *MPVue 4x4* é 78,4% no vetor de 128 pontos.

9.4.2 Filtro Passa-Baixas

No teste de filtro passa-baixas, uma imagem de 128KB – 362x362 pixels – é filtrada usando um filtro passa-baixas gaussiano com $\sigma = 1,0$ e são comparados os resultados de seis tamanhos de máscaras de filtragem: 3x3, 5x5, 7x7, 9x9, 11x11 e 13x13. A imagem de entrada é recebida e armazenada em 8 bits. Os valores intermediários dos cálculos, bem como os valores dos pesos dos filtros, são representados em ponto fixo de 32 bits, reservando 16 bits para a parte inteira e 16 bits para a parte fracionária. O resultado final é reconvertido para 8 bits para montar a imagem retornada.

Para distribuir as tarefas entre os processadores, a imagem é dividida em blocos de

Figura 33 – Eficiência das soluções no cálculo da FFT



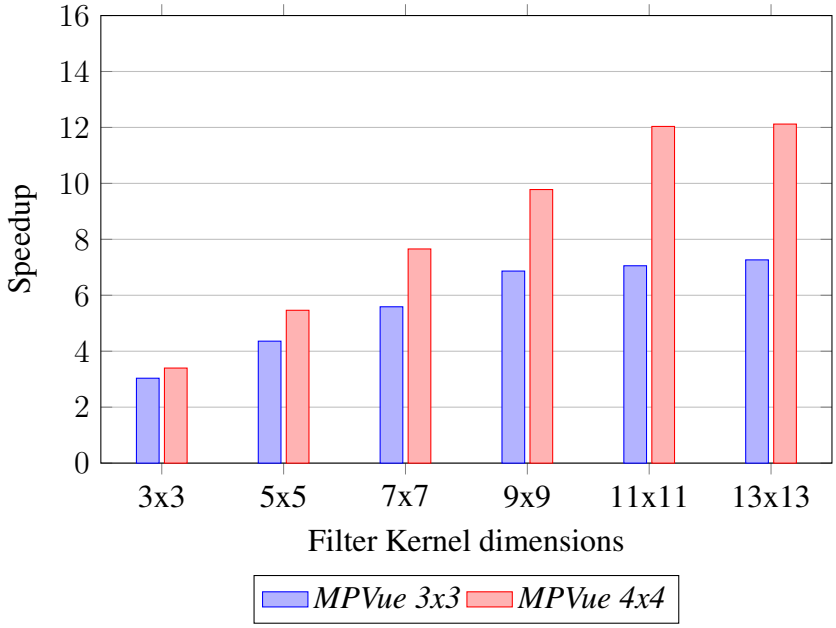
Fonte: do autor

tamanho 64x64. Uma sobreposição entre os blocos é realizada de forma a evitar artefatos resultantes das bordas inter-blocos. Cada LaPSIno é inicializado com os pesos do filtro a ser aplicado e recebe, iterativamente, um bloco para processar. Quando concluído, o bloco resultante é retornado ao LaPSIman. A imagem resultante é, então, reconstruída observando-se das sobreposições e bordas cuidadosamente. O processo é repetido até que não reste nenhum bloco faltando processar.

A Figura 34 mostra a aceleração para cada configuração comparado a uma única CPU, ou seja, quando o cálculo foi realizado apenas pelo LaPSIman. Uma vez que cada LaPSIno possui um limite no contexto de 4KB, cada lote de processamento é limitado a esse tamanho. Dessa forma, um bloco de imagem (64x64 pixels) é processado por vez. Para os menores filtros, nota-se proporcionalidade entre o valor da aceleração do sistema com o tamanho do filtro. Para filtro maior que 5x5, a aceleração é maior que 7x na configuração *MPVue 3x3*, atingindo 7,26x na janela 13x13. Já na configuração *MPVue 4x4*, o melhor desempenho é acima de 12x nos filtros maiores que 11x11, atingindo 12,12x na janela 13x13.

A comparação entre esse trabalho e as soluções recentes são apresentadas na Tabela 16. Ambas soluções utilizam a arquitetura CoreVA, e possuem um número diferente de elementos processadores. Por esse motivo, a comparação é realizada na forma da eficiência, conforme proposto por (AIROLDI; GARZIA; NURMI, 2010), ou seja, dividindo-

Figura 34 – Aceleração em cada configuração de fitro passa-baixas



Fonte: do autor

se a aceleração pelo número de processadores. Por serem soluções que contém *clusters*, mais de um core pode ser inserido em cada *tile* endereçado da NoC.

Tabela 16 – Comparativo de aceleração do filtro passa-baixas entre as soluções

	NoC	Núm *cores*	Aceleração	Eficiência
MPVue	3x3	9	7,26x	80,7%
	4x4	16	12,12x	75,8%
Sievers	2x2	16	4,5x	28,1%
	2x1	16	8,2x	51,3%
Ax	4x2	32	7,5x	23,4%
	2x2	32	10,5x	32,8%
	2x1	32	17,2x	53,8%

Primeiramente, Sievers (SIEVERS *et al.*, 2015) organiza 16 *cores* CoreVA em duas configurações de rede em chip: 2x2 e 2x1. Cada uma possui 4 e 8 processadores por *tile*, respectivamente. Seu melhor resultado, na configuração 2x2, é de 51,3%. Esse resultado é inferior a eficiências do *MPVue 4x4* (75,8%) e *MPVue 3x3* (80,7%).

Em segundo, Ax (AX *et al.*, 2018) organiza 32 *cores* CoreVA em três configurações de rede em chip: 4x2, 2x2 e 2x1; e compara utilizar um sistema de memória compartilhada com um sistema de memória distribuída (somente local). Uma vez que o MPVue utiliza o modelo de memória distribuída, a comparação será realizada com o modelo de memória distribuída de Ax. Apesar da eficiência ser melhor do que aquela mostrada por Sievers, o melhor resultado chegou em 53,8%.

9.5 Análise do Desempenho Paralelo

Nesse trabalho, para analisar o desempenho paralelo da execução do sistema, foi utilizado a técnica de *bloodgraph*, conforme apresentado na Seção 4.3. Dentro de um *bloodgraph* cada linha representa a ocupação de um processador ao longo do tempo. A largura da linha está em escala de forma a conter todo o tempo de processamento. Cada estado é representado por uma cor: cinza para *IDLE*, verde para *COMM*, azul para *TASK* e vermelho para *CONTROL*.

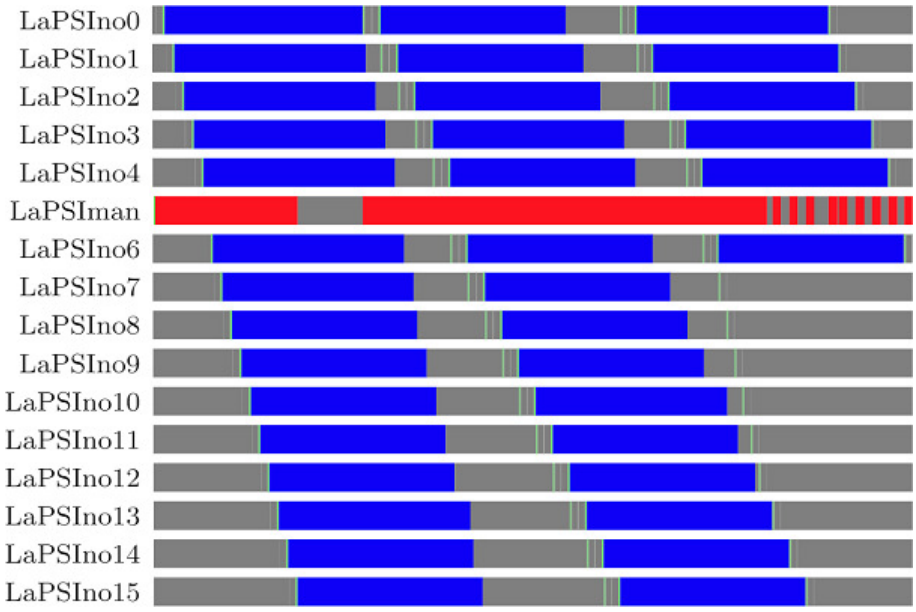
Duas execuções de filtro passa-baixas no *MPVue 4x4* foram escolhidas para se avaliar o efeito de perda de eficiência devido a código não-paralelizável: 5x5 e 11x11. A Figura 35 mostra a execução do filtro passa-baixas 5x5 nos processadores. Cada linha representa um processador localizado em um endereço da rede em chip, em ordem crescente. Portanto, a sexta linha representa o LaPSIman. Não há parte azul em sua linha porque ele não realiza cálculos nesse benchmark. Ao contrário, a parte vermelha representa o tempo em que estava preparando dados para serem enviados para os LaPSInos e reorganizando os dados recebidos dos mesmos.

É notado que o tempo em *IDLE* dos LaPSInos no início do processamento é devido ao tempo que o LaPSIman está preparando os dados a serem enviados, uma tarefa não paralelizável. O que pode ser reparado pelo fato do LaPSIman estar no estado *CONTROL*. As linhas verdes, que representam a comunicação, são visivelmente pequenas, mostrando uma baixa influência do tempo de comunicação no tempo geral do algoritmo. Como não há prioridade de recebimento de tarefas entre os LaPSInos, o LaPSIman distribui as tarefas sequencialmente no início do processo. Afinal, os LaPSInos iniciam o processamento imediatamente após receberem os dados a processar – representado pela linha azul logo após a verde.

Toda vez que um LaPSIno termina sua tarefa enquanto o LaPSIman está opcupado atendendo uma demanda de outro LaPSIno, ele entra em *IDLE*. Como, nessa execução, as tarefas de cada LaPSIno são relativamente rápidas, quando comparada com o tempo despendido pelo LaPSIman para preparar os dados, o LaPSIman tem pouco tempo para trocar os contextos e encaminhar uma nova tarefa. Isso redunda em espera por parte dos LaPSInos enquanto o LaPSIman encontra-se sobrecarregado, como pode ser visto em sua barra, predominantemente vermelha.

Ademais, no fim do processamento, alguns LaPSInos ficam em *IDLE* aguardando os outros processadores terminar suas tarefas. Isso representa uma perda de tempo de processamento indesejável. No entanto, é possível minimizar essa perda em ambientes que possuam múltiplas atividades a serem executadas. Uma nova atividade pode ser iniciada antes da atual ser concluída. Dessa forma, quando não há mais tarefas da atividade atual para serem encaminhadas aos LaPSInos, as tarefas da próxima atividade já podem ser atribuídas, desde que não dependam de resultados pendentes da atividade atual. Essa

Figura 35 – *Bloodgraph* de um filtro passa-baixas 5x5



Fonte: do autor

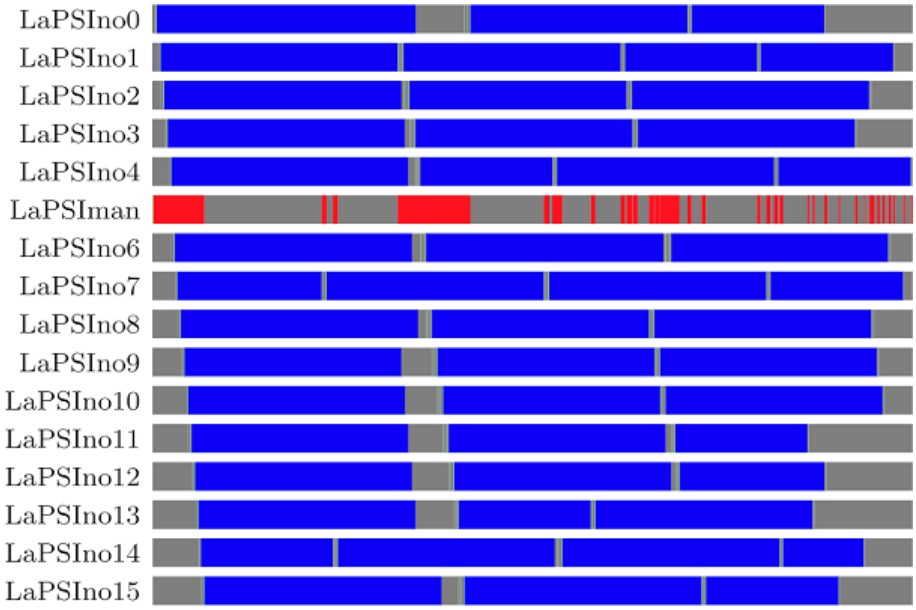
estratégia diminui o tempo em *IDLE* no início e no fim das atividades que foram concatenadas.

Por outro lado, a Figura 36 mostra o *bloodgraph* de uma execução do filtro passa-baixas 11x11. Da mesma forma que no caso anterior, cada linha representa a ocupação de cada LaPSIno, exceto a sexta linha, que representa o LaPSIman. Nessa situação, onde cada processamento leva mais tempo para ser executado, devido à maior quantidade de pesos no filtro, o efeito de sobrecarga no LaPSIman é menos notado.

Como cada LaPSIno demora mais para realizar sua tarefa, o LaPSIman consegue entregar todas as tarefas antes de ser requisitado novamente. Há uma pequena janela de ociosidade significativa desde o LaPSIno9 até o LaPSIno13 depois da execução da primeira tarefa. Isso se deve ao fato de haver coincidência de término do processamento de tarefas num intervalo muito curto, e não representa, necessariamente, que o LaPSIman esteja sobrecarregado. Seus efeitos de atraso nos outros LaPSInos não são notados no resto da execução, como ocorreu no caso anterior.

Os efeitos de assimetrias e preparação dos dados podem explicar também os resultados de cálculo da FFT na Seção 9.4. Na configuração *MPVue 3x3*, o melhor resultado foi no vetor de 512 pontos. Nesse tamanho de vetor, o sistema possui o maior ganho em paralelismo, uma vez que mais dados são processados por vez, otimizando o tempo gasto em preparação dos dados e comunicação. Todavia, na configuração *MPVue 4x4*, o tamanho do lote importa no tempo de execução. Os resultados mostram que, onde há uma assimetria na distribuição dos lotes entre os LaPSInos, o desempenho pode ser reduzido em até 20%. Isso pode ser explicado pelo fato de que quando não há lotes suficientes para todos os LaPSInos, alguns ficam ociosos, aguardando o processamento ser completado.

Figura 36 – *Bloodgraph* de um filtro passa-baixas 11x11



Fonte: do autor

O tempo necessário para preparar os dados e a comunicação entre os processadores, comparado com o tempo de computação, mostra que existe menos vantagem no processamento paralelo quando os dois primeiros possuem tempo de execução semelhante ao tempo de processamento. Na programação orientada a serviços, é crucial manter a harmonia entre o número e tamanho das tarefas de forma a obter o melhor desempenho. Disparar muitas tarefas rápidas, pode sobrecarregar o gerente, que tenta responder às muitas respostas dos processamentos que disparou. Por outro lado, tarefas grandes demais tornam o sistema suscetível a assimetrias na conclusão das tarefas, que resulta em tempo em *IDLE* – e, consequentemente, em perda de desempenho – tanto no gerente quando nos demais processadores.

9.6 Conclusão do Capítulo

Nesse capítulo foram apresentados os resultados da arquitetura proposta, comparados com trabalhos publicados recentemente. Foi demonstrado que a utilização combinada da arquitetura de hardware e de software pode resultar em um sistema que utiliza os recursos disponíveis de forma mais eficiente. Isso leva a melhores resultados em aceleração e eficiência, em todos as configurações comparáveis.

Juntos, a arquitetura proposta e o ambiente de desenvolvimento do MPVue, permitem uma pesquisa profunda nos gargalos do sistema e a proposição de soluções. Finalmente, o sistema está apto a implementar aplicações mais complexas, como as que serão vistas no próximo capítulo.

10 ESTUDO DE CASO

De forma a demonstrar o potencial da plataforma MPVue para desenvolver soluções de visão computacional, uma aplicação embarcada, chamada Pilgrim, é proposta. O algoritmo escolhido deve possuir complexidade computacional suficiente para que se possa fazer uma avaliação do desempenho da plataforma. Uma análise dos dados de desempenho mostrados na Seção 6.9, mostra que o algoritmo de Lucas-Kanade é responsável por grande parte do tempo gasto em computação. Então, a avaliação de uma solução embarcada que contemple sua execução é uma boa estimativa do comportamento do sistema para aplicações semelhantes. Para atingir esse objetivo, as potencialidades do sistema devem ser exploradas, bem como as suas restrições devem ser respeitadas. Dessa maneira, as funcionalidades escolhidas para estarem disponíveis foram implementadas como serviços, rodando a partir de contextos nos LaPSInos; a limitação de memória do sistema deve ser levada em consideração; as tarefas devem ser independentes em dados para que possam rodar em paralelo entre si.

O Capítulo é dividido em três partes. Na primeira, o desenvolvimento dos serviços disponíveis nos LaPSInos, chamado de Tek, é detalhado. Em seguida, a aplicação que roda no LaPSIman, chamada de Freyja, é apresentada. Por fim, os resultados da execução são discutidos.

10.1 Projeto Tek

O projeto Tek é formado pelo conjunto de serviços disponibilizados pelos LaPSInos para o LaPSIman dentro do Projeto Pilgrim. Para facilitar a manutenção e a execução do sistema, todos os LaPSInos contêm o mesmo software, disponibilizando, portanto, os mesmos serviços. Esse projeto é extremamente otimizado para que se consuma a menor quantidade de recursos e tempo executando as tarefas. Além disso, as tarefas são autônomas, ou seja, elas podem rodar somente com os dados disponíveis dentro da própria memória do processador, sem necessidade de acessos à memórias e periféricos externos, o que geraria latência. Uma vez que o contexto para uma tarefa está definido a execução começa e termina de forma muito rápida.

Devido ao tamanho da memória de dados do LaPSIno ser de 32KB, foi escolhido reservar 20KB para 5 contextos de 4KB cada. O restante da memória é utilizado com as variáveis de controle do sistema, as variáveis locais e a pilha, gerenciada pelo compilador.

No caso de um serviço que envolva processamento de imagens, o tamanho do bloco é limitado pelo contexto. No máximo, blocos de 64 x 64 pixels podem ser armazenados. Por isso, a quantidade de blocos e de *features* armazenados no processador é limitado.

Sempre que possível, os serviços utilizam números inteiros. Quando necessário, se utiliza uma implementação em ponto fixo de 32 bits (16 bits para parte inteira e 16 bits para parte fracionária). Apenas em último caso, para não se perder a precisão, se utiliza uma abordagem de ponto flutuante.

Além disso, um protocolo para comunicação entre o LaPSIman e os LaPSInos foi estabelecido. Ele segue o formato pergunta-resposta, onde uma pergunta sempre é seguida por uma resposta. A Tabela 17 mostra as funções implementadas e as condições de uso, bem como os dados enviados e a resposta esperada em cada comando.

Os serviços descritos na Tabela 17, podem ser separados em três grupos. Primeiro, os serviços de utilidade, que contemplam reset, ping e leitura e reinicio de contadores de desempenho. Eles não executam diretamente nenhuma atividade ligada à aplicação final, apenas servindo para funções acessórias que podem ser requisitadas durante a inicialização ou para manutenção do sistema. Depois, os serviços de transferência de dados, que contemplam leitura e escrita de atributos e dados. Eles servem como suporte à execução do algoritmo, para preparar os contextos ou para retornar os dados relativos a execução de outro serviço. Por fim, os serviços finalísticos, que executam funções relacionadas diretamente à atividade da aplicação. Gradientes, FFT, filtros e Lucas-Kanade são exemplos de serviços disponibilizados pelo projeto Tek.

Os serviços finalísticos normalmente recebem ponteiros para os contextos que já foram previamente estabelecidos. Nesses contextos estão os dados que devem ser operados e onde devem ser armazenados os resultados do algoritmo. Esses serviços normalmente retornam apenas um status, indicando o sucesso ou qual erro ocorreu durante sua execução.

10.1.1 Gradiente, FFT e Filtros

Os comandos de gradiente em x e em y são implementados de forma a apenas realizar a realizar a subtração dos pixels da imagem. O resultado do pixel atual é igual a subtração do próximo pixel pelo anterior, no eixo em que se pretende calcular. Os comandos de FFT e de filtro implementam as funções conforme foram descritas na Subseção 9.4.

10.1.2 Lucas-Kanade Embarcado

O desenvolvimento da função de Lucas-Kanade – conforme proposto por (BOUGUET, 2000) e descrita na Subseção 2.2.2 – no âmbito do Tek teve que ser realizada

Tabela 17 – Serviços implementados no projeto Tek

Serviço	Comando		Resposta	
Reset	[0]	comando `reset`	[0]	status
Ping	[0]	comando `ping`	[0]	pong
Ler atributos	[0]	comando `le_atrib`	[0]	status
	[1]	contexto origem	[1]	dados[...]
Escrever atributos	[0]	comando `escreve_atrib`	[0]	status
	[1]	contexto destino		
	[2:]	dados[...]		
Ler dados	[0]	comando `le_dados`	[0]	status
	[1]	contexto origem	[1]	dados[...]
	[2]	offset		
	[3]	tamanho		
Escrever dados	[0]	comando `escreve_dados`	[0]	status
	[1]	contexto destino		
	[2]	offset		
	[3]	tamanho		
	[4:]	dados[...]		
Gradiente em x	[0]	comando `grad_x`	[0]	status
	[1]	contexto imagem destino		
	[2]	contexto imagem origem		
Gradiente em y	[0]	comando `grad_y`	[0]	status
	[1]	contexto imagem destino		
	[2]	contexto imagem origem		
FFT	[0]	comando `fft`	[0]	status
	[1]	contexto parte real destino		
	[2]	contexto parte imag. destino		
	[3]	contexto parte real origem		
	[4]	contexto parte imag. origem		
Filtro	[0]	comando `filtro`	[0]	status
	[1]	opções		
	[2]	contexto imagem destino		
	[3]	contexto imagem origem		
	[4]	contexto pesos filtro		
Lucas-Kanade	[0]	comando `lk`	[0]	status
	[1]	contexto imagem atual		
	[2]	contexto imagem anterior		
	[3]	contexto *features* atual		
	[4]	contexto *features* anterior		
	[5]	contexto resultado		
Ler cont. de desempenho	[0]	comando `le_perf`	[0]	status
			[1]	dados[...]
Reiniciar cont. desempenho	[0]	comando `reset_perf`	[0]	status
	[1]	novo valor		

tomando alguns cuidados. Os valores das imagens, das pirâmides e seus gradientes foram calculados utilizando apenas números inteiros. Os pontos a serem rastreados são calculados como números de ponto fixo de 32 bits, com 16 bits para a parte real e 16 bits para a parte imaginária. Somente a última etapa do algoritmo, onde é feito o cálculo da diferença, foi utilizada a biblioteca de ponto flutuante do compilador.

Dessa forma, quando utiliza-se a Equação (13) durante a execução dos algoritmos, os valores de G e $\bar{b}$ são inteiros. Para manter a precisão, sem correr risco de *underflow* ou *overflow* no cálculo de G^{-1}, a mesma é calculada em ponto flutuante. O passo da iteração ν resulta em ponto flutuante e é convertido de volta a ponto fixo para ser somando ao ponto rastreado.

Assim, quatro divisões e quatro multiplicações são executadas em ponto flutuante por iteração por ponto. Apesar de aumentar o tempo de computação, foi priorizada a precisão do resultado nessa parte crítica.

Os parâmetros do algoritmo foram selecionados empiricamente. Foi utilizada uma janela quadrada de 11 pixels de lado. O ponto é considerado encontrado quando em uma iteração o valor de atualização é menor do que 0,03 pixels em x e em y. Se 5 iterações se passaram sem convergência, o ponto é considerado não encontrado.

10.2 Projeto Freyja

O projeto Freyja corresponde à aplicação mestre do MPVue para execução das funcionalidades de visão computacional. Ela é responsável por receber os dados através da entrada de dados, dividi-los em blocos e transmitir aos LaPSInos, conforme sua afinidade, além de enviar os comandos de processamento, recebendo, após os metadados gerados, armazenado-os e fornecendo as informações relevantes aos dispositivos conectados no sistema.

O conjunto formado por um serviço finalístico e os dados exigidos para sua execução é chamado *job*. Para executar um *job* pode ser necessário a requisição de serviços de suporte, a fim de disponibilizar os dados nos contextos corretos.

Para a execução da aplicação, são gerados *jobs* que ficam enfileirados esperando execução. A cada iteração, o agendador de tarefas verifica qual LaPSIno está disponível. À ele é designado um *job*, baseado em critério de minimizar a transferência de dados. Ou seja, um *job* é preferido sobre outro se os seus dados de entrada já estão disponíveis no LaPSIno. Quando o processamento é concluído, o *job* é marcado como executado e os dados pertinentes são retornados. O LaPSIno é colocado novamente como disponível para uma nova execução.

O fluxograma do algoritmo foi inspirado no protótipo implementado, apresentado no Capítulo 5. Entretanto, para poder rodar embarcado, dentro das limitações do hardware, a aplicação teve de ser toda reimplementada em código C e alterada para utilizar alocação

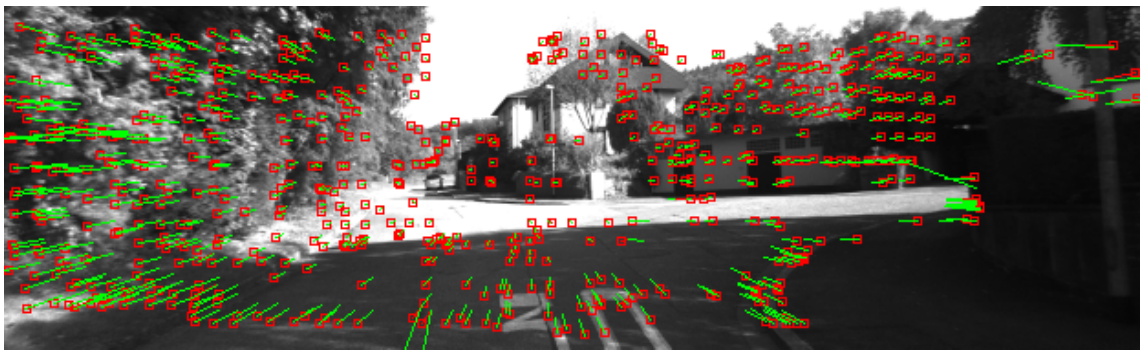
estática de memória. A funcionalidade de Lucas-Kanade foi disponibilizada através de um comando serial que recebia uma imagem e retornava os pontos rastreados.

10.3 Resultados

O projeto Pilgrim, contendo ambos os projetos Tek e Freyja, foi executado no ambiente contendo o MPVue instanciado na placa KC705, rodando a 50MHz. As imagens foram enviadas ao sistema utilizando a comunicação serial, através de um script Python que as lia sequencialmente e as enviava uma a uma ao sistema. O resultado – as posições das *features* rastreadas – era lido através da serial e indicava a conclusão do processamento da imagem.

Para esse teste foram utilizadas as imagens do dataset KITTI, o mesmo utilizado para validar o protótipo em Python. Os resultados podem ser observados na Figura 37, onde nota-se que o sistema conseguiu com alto grau de sucesso rastrear as *features* entre dois *frames*.

Figura 37 – Resultado do algoritmo de Lucas-Kanade implementado pelo Pilgrim, rodando no MPVue. Dataset: KITTI, sequência 3, *frames* 246 e 247



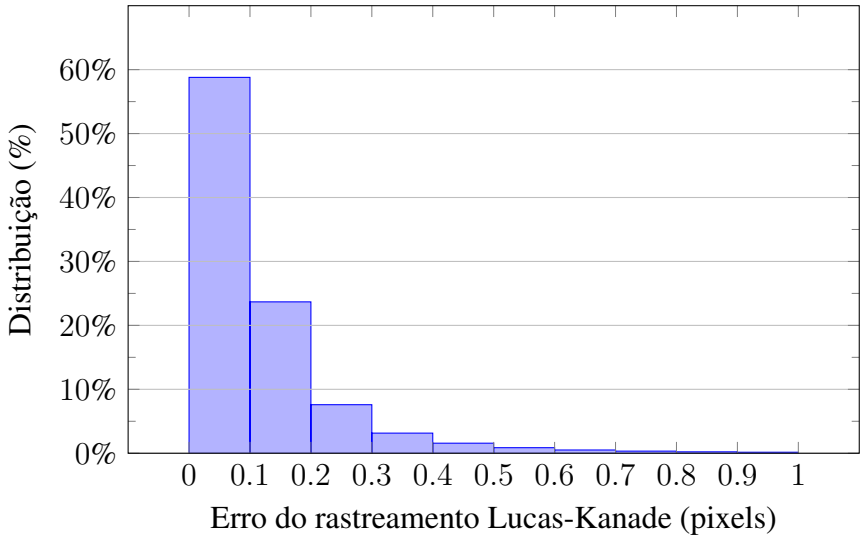
Fonte: do autor

10.3.1 Precisão

Com o objetivo de avaliar a precisão do algoritmo de Lucas-Kanade implementado no Pilgrim, seus resultados foram comparados com os obtidos da função de referência, proposta por (BOUGUET, 2000) e disponibilizada no OpenCV (BRADSKI, 2000). Então, a distância euclidiana entre o valor retornado pelo Pilgrim e o valor retornado pelo OpenCV foram obtidos e analisados. A Figura 38 mostra o histograma dos valores dessas distâncias para a sequencia 4. É possível notar que 58,8% dos valores possuem distância entre 0,0 e 0,1. Ademais, a quantidade de pontos cuja estimativa realizada pelo Pilgrim possui distância da estimativa da OpenCV maior do que 1 pixel é de 3%. Isso demonstra que os valores obtidos pelo algoritmo, apesar de rodarem utilizando números inteiros e com ponto fixo em quase toda a execução, possui precisão muito próxima do algoritmo

de referência.

Figura 38 – Histograma das distâncias dos valores obtidos no rastreamento pelo Pilgrim e o OpenCV. Dataset: KITTI, sequência 3



Fonte: do autor

10.3.2 Taxa de Pontos Encontrados

Outra avaliação pertinente realizada é se avaliar a quantidade de pontos encontrados nas duas soluções. A Tabela 18 mostra a taxa de pontos encontrados comparado com o total de pontos que foram selecionados. Os dados foram extraídos durante uma rodada na sequencia 3, por todos os frames. 852640 pontos foram selecionados no total para serem rastreados em 801 *frames*.

Tabela 18 – Comparativo de taxa de pontos rastreados, comparados com o total de pontos selecionados. Dataset: KITTI, sequência 3

Solução	Pontos	(%)
Total	852640	100,00%
OpenCV	797976	93,59%
Pilgrim	470613	55,19%

Enquanto a implementação do Lucas-Kanade presente na OpenCV foi capaz de encontrar os pontos em 93,59% das vezes, a implementação realizada no projeto Pilgrim conseguiu encontrar os pontos em apenas 55,19% das vezes. Isso pode ser explicado por alguns fatores inerentes do formato escolhido. Se uma *feature* possui deslocamento maior que o limite do bloco, ela é considerada perdida. Mesmo que exista superposição de blocos, o movimento é limitado a um valor menor que o tamanho do bloco. Além disso, o OpenCV utiliza pontos flutuantes para todos os cálculos, enquanto o Pilgrim utiliza inteiros e pontos fixos. Em regiões de baixo gradiente, os valores calculados podem ser

truncados, o que pode levar a um gradiente pouco expressivo, prejudicando a convergência.

10.3.3 Tempo de Execução

O tempo de execução do algoritmo de Lucas-Kanade do Pilgrim é mostrado na Tabela 19. Essa tabela contém os dados em cada *frame* da quantidade de pontos rastreados e a quantidade de pontos encontrados, bem como o tempo gasto com a execução do algoritmo. Pode-se notar que o tempo médio gasto para rastrear 1801 pontos é de 0,628s.

Tabela 19 – Tempo de Execução por *frame*. Dataset: KITTI, sequência 3, *frames* 1 a 29

frame	Pontos Rastreados	Pontos Encontrados	Tempo(s)
00002	1738	570	0,605
00003	1794	690	0,617
00004	1779	754	0,633
00005	1762	754	0,611
00006	1798	806	0,631
00007	1857	833	0,619
00008	1849	920	0,650
00009	1864	920	0,611
00010	1793	913	0,652
00011	1825	833	0,627
00012	1806	910	0,620
00013	1751	871	0,608
00014	1724	859	0,649
00015	1779	866	0,618
00016	1837	1037	0,625
00017	1815	1030	0,624
00018	1907	995	0,649
00019	1780	1011	0,654
00020	1803	938	0,663
00021	1800	919	0,642
00022	1753	888	0,624
00023	1871	911	0,618
00024	1860	963	0,641
00025	1754	937	0,623
00026	1827	850	0,630
00027	1776	752	0,621
00028	1810	769	0,620
00029	1718	758	0,599

Esse tempo é insuficiente para execução durante a aquisição das imagens – que ocorre a cada 0,1s – na forma atual do sistema. Todavia, cabe lembrar que, nesse teste, o sistema operava a 50MHz, uma fração das frequências atingidas por circuitos modernos.

Apesar da limitação de frequência encontrada na plataforma PULPino quando exe-

cutando em FPGA, esses mesmos sistemas quando implementados em ASIC possuem limite de frequência de ordem de grandeza maior do que o primeiro. Por exemplo, o projeto Quentin (SCHIAVONE *et al.*, 2018), que utiliza o mesmo processador RI5CY que o MPVue, foi produzido em 22nm e atingiu 650MHz. Assim sendo, ao se considerar que o MPVue em conjunto com o Pilgrim leva aproximadamente 0,6s para executar 1801 pontos a 50MHz, obtém-se o valor de 17500 ciclos de clock por ponto. Se o sistema rodasse a 325MHz, metade da frequência obtida por (SCHIAVONE *et al.*, 2018) já seria suficiente para executar o algoritmo mais rápido que o tempo de aquisição entre *frames* (0,1s).

10.4 Conclusão do Capítulo

Nesse capítulo, foi analisada uma aplicação para o MPVue, que implementou o método de Lucas-Kanade. O desempenho desse programa serviram para avaliar a capacidade da plataforma de executar algoritmos de visão computacional. Os resultados mostraram que o MPVue é adequado para esse tipo de aplicação, obtendo desempenho compatível com os algoritmos do estado da arte. Isso não quer dizer que não há melhorias a serem realizadas, como na taxa de pontos encontrados e na frequência máxima de operação do sistema, mas elas estão mapeadas e podem ser desenvolvidas em um trabalho futuro.

11 CONCLUSÃO

Esse trabalho apresentou uma nova arquitetura de processamento paralelo para visão computacional, chamada MPVue. Ela é preparada para executar os principais algoritmos de visão computacional, que formam a base das soluções de mapeamento do ambiente. Considerando os avanços recentes na área de processamento de imagens em sistemas embarcados, os algoritmos alvo dessa aplicação foram analisados, principalmente no que tange à escolha e o rastreamento de *features*, além dos algoritmos que fazem parte do ecossistema de soluções reconstrução 3D: Estrutura a Partir do Movimento, Localização e Mapeamento Simultâneo e Odometria Visual.

A plataforma proposta busca uma sinergia entre hardware e software direcionada à otimização do processo de visão computacional. Os algoritmos a serem implementados passaram por uma etapa de *profiling*, para determinar quais precisariam ser submetidos a um processo de otimização para o sistema.

As arquiteturas MPSoC desenvolvidas são configuráveis e podem ser definidas em tempo de projeto para responder às necessidades da aplicação. Processadores ou outros elementos de hardware, como memórias e aceleradores, podem ser adicionados ou removidos da rede previamente dependendo da aplicação desejada.

A partir desse ponto, se decidiu pela implementação em software de um algoritmo muito utilizado para cálculo de fluxo óptico: o Lucas-Kanade.

No Capítulo 4 as diversas plataformas de desenvolvimento utilizadas durante a tese foram apresentadas. Além disso, as formas de verificação de desempenho e as formas de comparação com outras soluções do estado da arte foram demonstradas.

No Capítulo 5 foi discutida a forma como o sistema deveria funcionar para solucionar o problema de reconstrução 3D em um sistema embarcado. A fim de tornar o algoritmo passível de ser utilizado em tais condições, considerando as restrições de tais sistemas, um novo método de cálculo de profundidade foi proposto. Tal método foi aplicado pois, além de servir para o referido cálculo, as informações de ângulo de paralaxe dele resultantes também poderão ser utilizadas como medida de erro. Assim, a medida em questão poderá ser analisada em um trabalho futuro, para verificar zonas da imagem onde o fluxo óptico possui direção e magnitude discrepante com o resto da cena, o que pode levar à

descoberta de outros objetos de interesse, principalmente aqueles não estáticos em relação ao ambiente. Também, o fluxograma da solução foi apresentado.

No Capítulo 6 foram mostrados os algoritmos e os detalhes de implementação do protótipo desenvolvido em linguagem Python para testar a funcionalidade e a viabilidade do sistema. Ademais, definições de algoritmos e estrutura de suporte foram apresentadas para tornar possível a implementação do sistema final em FPGA. Uma análise de erros é realizada, mostrando uma forma de filtragem eficaz para eliminar pontos errados do cálculo, antes que eles poluam a nuvem de pontos gerada. Os resultados intermediários, do algoritmo rodando em PC, foram apresentados e mostraram resultados promissores.

A plataforma foi detalhada, mostrando o desenvolvimento de sistemas multiprocessados interligados por rede em chip. Dois processadores (LaPSIman e LaPSIno) foram propostos e desenvolvidos utilizando como base a plataforma PULPino (TRABER *et al.*, 2016).

A arquitetura de software embarcado traz o conceito de Arquitetura Orientada a Serviços. Para ser possível implementar essa arquitetura, utilizou-se o modelo de programação orientada a serviços. A arquitetura de software foi montada em um modelo *bottom-up*, desde as funções de baixo nível até os módulos ligados a parte importante do sistema.

Os resultados de *benchmark*, baseados no StreamIt (THIES; KARCZMAREK; AMARASINGHE, 2002) (FFT e filtro passa-baixas) foram superiores aos encontrados no estado da arte. Essa análise gerou o artigo [1] já submetido à publicação em periódico. Além disso, a ferramenta de análise utilizando *bloodgraphs* é importante para identificar gargalos na execução dos algoritmos. Foi demonstrado que uma boa relação entre a quantidade de dados transmitidos e o processamento realizado é determinante para evitar gargalos no gerente durante a execução dos processos.

O estudo de caso, contendo uma aplicação que roda o algoritmo de Lucas-Kanade, obteve resultados comparáveis aos obtidos na biblioteca OpenCV (BRADSKI, 2000), considerada o estado da arte em processamento de imagens. Essa análise comprovou que a plataforma permite do desenvolvimento de sistemas eficientes, fornecendo as melhores técnicas da literatura, aliada com as ferramentas corretas para medir e controlar o desempenho do sistema em diferentes aspectos.

Observou-se, ao final da tese e da implementação do sistema proposto, no entanto, que ainda existem muitos tempos que podem ser melhorados. A implementação de periféricos de otimização de trabalhos repetitivos, como DMA e aceleradores específicos de hardware pode ser incorporada ao sistema, potencializando as aplicações da plataforma. Um sistema operacional deve ser incorporado, para aprimorar o agendamento de tarefas. Além disso, há bastante espaço para aperfeiçoar as interfaces do sistema, agregando periféricos de alta velocidade para a comunicação com outros sistemas.

[1] ILHA, G.et al.MPVue – A Multi-Processor System-on-Chip Platform forComputer Vision. submetido à publicação. 2019

Por fim, conclui-se que a pesquisa e os trabalhos desenvolvidos na presente tese devem permanecer em constante aprimoramento, de modo a permitir a busca de outras aplicações que possam ser implementadas na plataforma. Citam-se como exemplo principal aquelas que se beneficiem de processamento paralelo, como algoritmos de processamento de imagens para implementação de soluções em visão computacional voltadas ao auxílio ao motorista, detecção de obstáculos, redes neurais para identificação de objetos e soluções de auxílio a diagnósticos médicos, temas extremamente atuais e com alto potencial frente ao mercado e às pesquisas acadêmicas de desenvolvimento de soluções desta natureza.

REFERÊNCIAS

AGUILAR-GONZÁLEZ, A.; ARIAS-ESTRADA, M.; BERRY, F. Robust feature extraction algorithm suitable for real-time embedded applications. **Journal of Real-Time Image Processing**, [S.l.], v.14, n.3, p.647–665, Mar. 2018.

AIROLDI, R.; GARZIA, F.; NURMI, J. FFT algorithms evaluation on a homogeneous multi-processor system-on-chip. *In*: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING WORKSHOPS, 39., 2010, San Diego, CA, USA. **Proceedings [...]** [S.l.: s.n.], 2010. p.58–64.

Arm Ltd. **AMBA Specification Rev 2.0 | AMBA Specification Rev 2.0**. [S.l.]: Arm Ltd, 1999. Disponível em: http://www.arm.com. Acesso em: 09 maio 2019.

AX, J. *et al.* CoreVA-MPSoC: a many-core architecture with tightly coupled shared and local data memories. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.29, n.5, p.1030–1043, May 2018.

AYDI, Y. *et al.* Design and performance evaluation of a reconfigurable Delta MIN for MPSOC. *In*: INTERNATONAL CONFERENCE ON MICROELECTRONICS, 2007, Cairo, Egypt. **Proceedings [...]** [S.l.: s.n.], 2007. p.115–118.

BAHN, J. H. *et al.* On design and application mapping of a network-on-chip(NoC) architecture. **Parallel Processing Letters**, [S.l.], v.18, n.02, p.239–255, June 2008.

BAHN, J. H.; YANG, J.; BAGHERZADEH, N. Parallel FFT algorithms on network-on-chips. *In*: INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: NEW GENERATIONS (ITNG 2008), 5., 2008, Las Vegas, NV, USA. **Proceedings [...]** [S.l.: s.n.], 2008. p.1087–1093.

BAY, H.; TUYTELAARS, T.; VAN GOOL, L. SURF: speeded up robust features. *In*: COMPUTER VISION – ECCV 2006, 2006, Graz, Austria. **Proceedings [...]** Springer Berlin Heidelberg, 2006. p.404–417. (Lecture Notes in Computer Science).

BENINI, L.; MICHELI, G. D. Networks on chips: a new SoC paradigm. **Computer**, [S.l.], v.35, n.1, p.70–78, Jan. 2002.

BINESH, M. M. **An Analysis of NoCs in FPGAs**. 2013. Tese (Doutorado em Engenharia Elétrica) — McMaster University, Hamilton, Ontario, Canada, 2013.

BINH, H. T. T.; LOI, M. D.; THUY, N. T. Improving image segmentation using genetic algorithm. *In*: INTERNATIONAL CONFERENCE ON MACHINE LEARNING AND APPLICATIONS, 11., 2012, Boca Raton, FL, USA. **Proceedings [...]** IEEE, 2012. v.2, p.18–23.

BORENSTEIN, E.; ULLMAN, S. Combined top-down/bottom-up segmentation. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, [S.l.], v.30, n.12, p.2109–2125, Dec. 2008.

BOTERO-GALEANO, D.; GONZALEZ, A.; DEVY, M. Architecture embarquée pour le SLAM monoculaire. *In*: RFIA 2012 (RECONNAISSANCE DES FORMES ET INTELLIGENCE ARTIFICIELLE), 2012, Lyon, France. **Proceedings [...]** [S.l.: s.n.], 2012. p.978–2–9539515–2–3.

BOUGUET, J.-y. Pyramidal implementation of the Lucas Kanade feature tracker. **Intel Corporation, Microprocessor Research Labs**, [S.l.], 2000.

BRADSKI, G. **The OpenCV Library**. 2000. Disponível em: http://www.drdobbs.com/open-source/the-opencv-library/184404319. Acesso em: 28 maio 2019.

BRENOT, F.; PIAT, J.; FILLATREAU, P. FPGA based hardware acceleration of a BRIEF correlator module for a monocular SLAM application. *In*: INTERNATIONAL CONFERENCE ON DISTRIBUTED SMART CAMERA, 10., 2016, New York, NY, USA. **Proceedings [...]** ACM, 2016. p.184–189. (ICDSC '16).

BROCKETT, R. W. Robotic manipulators and the product of exponentials formula. *In*: MATHEMATICAL THEORY OF NETWORKS AND SYSTEMS, 1984, Beer Sheva, Israel. **Proceedings [...]** Springer Berlin Heidelberg, 1984. p.120–129. (Lecture Notes in Control and Information Sciences).

CALONDER, M. *et al.* BRIEF: binary robust independent elementary features. *In*: COMPUTER VISION – ECCV 2010, 2010, Heraklion, Crete, Greece. **Proceedings [...]** Springer Berlin Heidelberg, 2010. p.778–792. (Lecture Notes in Computer Science).

CANNY, J. A computational approach to edge detection. *In*: FISCHLER, M. A.; FIRSCHEIN, O. (ed.). **Readings in computer vision**. San Francisco (CA): Morgan Kaufmann, 1987. p.184–203.

CASTRILLON, J. *et al.* A hardware/software stack for heterogeneous systems. **IEEE Transactions on Multi-Scale Computing Systems**, [S.l.], v.4, n.3, p.243–259, July 2018.

CENTIR, M. *et al.* A combined color-correlation visual model for object tracking using particle filters. *In*: INTERNATIONAL SYMPOSIUM ON IMAGE AND SIGNAL PROCESSING AND ANALYSIS (ISPA), 2013., 2013, Trieste, Italy. **Proceedings [...]** [S.l.: s.n.], 2013. p.153–158.

CHIUSO, A.; BROCKETT, R.; SOATTO, S. Optimal structure from motion: local ambiguities and global estimates. **International Journal of Computer Vision**, [S.l.], v.39, n.3, p.195–228, Sept. 2000.

CIVERA, J.; DAVISON, A. J.; MONTIEL, J. M. Inverse depth parametrization for monocular SLAM. **IEEE transactions on robotics**, [S.l.], v.24, n.5, p.932–945, 2008.

CIVERA, J. *et al.* 1-Point RANSAC for extended Kalman filtering: application to real-time structure from motion and visual odometry. **Journal of Field Robotics**, [S.l.], v.27, n.5, p.609–631, 2010.

CONTI, F. *et al.* Energy-efficient vision on the PULP platform for ultra-low power parallel computing. *In*: IEEE WORKSHOP ON SIGNAL PROCESSING SYSTEMS (SiPS), 2014, Belfast, UK. **Proceedings [...]** [S.l.: s.n.], 2014. p.1–6.

CRANDALL, D. *et al.* Discrete-continuous optimization for large-scale structure from motion. *In*: CVPR 2011, 2011, Colorado Springs, CO, USA. **Proceedings [...]** [S.l.: s.n.], 2011. p.3001–3008.

DAS, A. K. *et al.* **Reliable and Energy Efficient Streaming Multiprocessor Systems**. 1.ed. [S.l.]: Springer, 2018. (Embedded Systems).

DAVIDSON, S. *et al.* The Celerity open-source 511-core RISC-V tiered accelerator fabric: fast architectures and design methodologies for fast chips. **IEEE Micro**, [S.l.], v.38, n.2, p.30–41, Mar. 2018.

DAVISON, A. J. *et al.* MonoSLAM: real-time single camera slam. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, [S.l.], v.29, n.6, p.1052–1067, June 2007.

DINECHIN, B. D. d. *et al.* A clustered manycore processor architecture for embedded and accelerated applications. *In*: IEEE HIGH PERFORMANCE EXTREME COMPUTING CONFERENCE (HPEC), 2013, Waltham, MA, USA. **Proceedings [...]** [S.l.: s.n.], 2013. p.1–6.

DOGAN, A. Y. *et al.* Multi-core architecture design for ultra-low-power wearable health monitoring systems. *In*: DESIGN, AUTOMATION TEST IN EUROPE CONFERENCE EXHIBITION (DATE), 2012, Dresden, Germany. **Proceedings [...]** [S.l.: s.n.], 2012. p.988–993.

DUVALL, M. **Spatial software pipelining on distributed architectures for sparse matrix codes**. 2004. Tese (Doutorado em Engenharia Elétrica) — Massachusetts Institute of Technology, Boston, 2004.

EADE, E.; DRUMMOND, T. Scalable monocular SLAM. *In*: IEEE COMPUTER SOCIETY CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION - VOLUME 1, 2006, Washington, DC, USA. **Proceedings [...]** IEEE Computer Society, 2006. p.469–476. (CVPR '06).

ELMOHR, M. A. *et al.* RVNoC: a framework for generating RISC-V NoC-based MPSoC. *In*: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 26., 2018, Cambridge, UK. **Proceedings [...]** [S.l.: s.n.], 2018. p.617–621.

EMER, J. S.; CLARK, D. W. A characterization of processor performance in the Vax-11/780. *In*: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 11., 1984, New York, NY, USA. **Proceedings [...]** ACM, 1984. p.301–310. (ISCA '84).

ENGEL, J.; SCHöPS, T.; CREMERS, D. LSD-SLAM: Large-scale direct monocular SLAM. *In*: EUROPEAN CONFERENCE ON COMPUTER VISION, 2014, Zurich, Switzerland. **Proceedings [...]** Springer, 2014. p.834–849.

ENGEL, J.; STURM, J.; CREMERS, D. Semi-dense visual odometry for a monocular camera. *In*: IEEE INTERNATIONAL CONFERENCE ON COMPUTER VISION, 2013, Sydney, NSW, Australia. **Proceedings [...]** IEEEE, 2013. p.1449–1456.

ENGEL, J.; STüCKLER, J.; CREMERS, D. Large-scale direct SLAM with stereo cameras. *In*: IEEE/RSJ INTERNATIONAL CONFERENCE ON INTELLIGENT ROBOTS AND SYSTEMS (IROS), 2015, Hamburg, Germany. **Proceedings [...]** [S.l.: s.n.], 2015. p.1935–1942.

ETH Zurich. **PULP platform**. 2019. Disponível em: https://pulp-platform.org/. Acesso em: 09 fev. 2019.

FERNANDEZ-ALONSO, E. *et al.* A NoC-based multi-softcore with 16 cores. *In*: IEEE INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS AND SYSTEMS, 17., 2010, Athens, Greece. **Proceedings [...]** [S.l.: s.n.], 2010. p.259–262.

FIELDS, J. *et al.* Monocular structure from motion for near to long ranges. *In*: IEEE INTERNATIONAL CONFERENCE ON COMPUTER VISION WORKSHOPS, ICCV WORKSHOPS, 12., 2009, Kyoto, Japan. **Proceedings [...]** [S.l.: s.n.], 2009. p.1702–1709.

GAO, X.-S. *et al.* Complete solution classification for the perspective-three-point problem. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, [S.l.], v.25, n.8, p.930–943, Aug. 2003.

GARIBOTTI, R. *et al.* Simultaneous multithreading support in embedded distributed memory MPSoCs. *In*: ACM/EDAC/IEEE DESIGN AUTOMATION CONFERENCE (DAC), 50., 2013, Austin, TX, USA. **Proceedings [...]** [S.l.: s.n.], 2013. p.1–7.

GEIGER, A.; LENZ, P.; URTASUN, R. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. *In*: CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR), 2012, Providence, RI, USA. **Proceedings [...]** [S.l.: s.n.], 2012.

GIRARDEAU-MONTAUT, D. **CloudCompare - Open Source project**. 2011. Disponível em: https://cloudcompare.org/. Acesso em: 18 maio 2019.

HARRIS, C.; STEPHENS, M. A combined corner and edge detector. *In*: ALVEY VISION CONFERENCE, 1988, Alvey, UK. **Proceedings [...]** Alvety Vision Club, 1988. p.23.1–23.6.

HEEGER, D. J.; JEPSON, A. D. Subspace methods for recovering rigid motion I: Algorithm and implementation. **International Journal of Computer Vision**, [S.l.], v.7, n.2, p.95–117, Jan. 1992.

HEENE, M. R. *et al.* **Single chip microcomputer with patching and configuration controlled by on-board non-volatile memory**. 1989. n.US4802119A. Disponível em: https://patents.google.com/patent/US4802119A/en. Acesso em: 02 mar. 2019.

HORAUD, R. *et al.* An analytic solution for the perspective 4-point problem. *In*: CVPR '89: IEEE COMPUTER SOCIETY CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, 1989, San Diego, CA, USA. **Proceedings [...]** [S.l.: s.n.], 1989. p.500–507.

HUANG, G.; MOURIKIS, A.; ROUMELIOTIS, S. **Genearalized analysis and improvement of the consistency for EKF-based SLAM**. Minneapolis, MN: University of Minnesota, 2008. Technical report.

HUANG, G. P.; MOURIKIS, A. I.; ROUMELIOTIS, S. I. A first-estimates Jacobian EKF for improving SLAM consistency. *In*: KHATIB, O.; KUMAR, V.; PAPPAS,

G. J. (ed.). **Eleventh International Symposium on Experimental Robotics**. Berlin, Heidelberg: Springer, 2009. p.373–382.

HUNT, M.; ROWSON, J. A. Blocking in a system on a chip. **IEEE Spectrum**, [S.l.], v.33, n.11, p.35–41, Nov. 1996.

IONESCU, L. **The RISC-V Embedded GCC**. 2017. Disponível em: https://gnu-mcu-eclipse.github.io/toolchain/riscv/. Acesso em: 10 fev. 2019.

JOHNSSON, O.; STENEMO, M.; ABDIN, Z. ul. **Programming & implementation of streaming applications**. 2005. Dissertação (Mestrado em Engenharia Elétrica) — Högskolan i Halmstad/Sektionen för Informationsvetenskap, Data- och Elektroteknik (IDE), 2005.

JOVEN, J. *et al.* xENoC - An experimental network-on-chip environment for parallel distributed computing on NoC-based MPSoC architectures. *In*: EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP 2008), 16., 2008, Toulouse, France. **Proceedings [...]** IEEE, 2008. p.141–148.

KHAMIS, M. *et al.* A configurable RISC-V for NoC-based MPSoCs: a framework for hardware emulation. *In*: INTERNATIONAL WORKSHOP ON NETWORK ON CHIP ARCHITECTURES (NoCArc), 11., 2018, Fukuoka, Japan. **Proceedings [...]** IEEE, 2018. p.1–6.

KLEIN, G.; MURRAY, D. Parallel tracking and mapping for small AR workspaces. *In*: IEEE AND ACM INTERNATIONAL SYMPOSIUM ON MIXED AND AUGMENTED REALITY, 6., 2007, Washington, DC, USA. **Proceedings [...]** IEEE Computer Society, 2007. p.1–10. (ISMAR '07).

LEE, M.-J.; JUNG, Y.-J. FPGA implementation of Levenverg-Marquardt algorithm. **Journal of the Institute of Electronics and Information Engineers**, [S.l.], v.51, n.11, p.73–82, 2014.

LEPETIT, V.; MORENO-NOGUER, F.; FUA, P. EPnP: An accurate O(n) solution to the PnP problem. **International Journal of Computer Vision**, [S.l.], v.81, n.2, p.155, July 2008.

LIM, H.; LIM, J.; KIM, H. J. Real-time 6-DOF monocular visual SLAM in a large-scale environment. *In*: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION (ICRA), 2014, Hong Kong, China. **Proceedings [...]** IEEE, 2014. p.1532–1539.

LONGUET-HIGGINS, H. C. A computer algorithm for reconstructing a scene from two projections. **Nature**, [S.l.], v.293, n.5828, p.133, Sept. 1981.

LOWE, D. G. Distinctive Image Features from Scale-Invariant Keypoints. **International Journal of Computer Vision**, [S.l.], v.60, n.2, p.91–110, Nov. 2004.

LUCAS, B. D.; KANADE, T. An iterative image registration technique with an application to stereo vision. *In*: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE - VOLUME 2, 7., 1981, Vancouver, BC, Canada. **Proceedings [...]** Morgan Kaufmann, 1981. p.674–679. (IJCAI'81).

MARONGIU, A.; BURGIO, P.; BENINI, L. Supporting OpenMP on a multi-cluster embedded MPSoC. **Microprocessors and Microsystems**, [S.l.], v.35, n.8, p.668–682, Nov. 2011.

MEDATHATI, N. V. K. *et al.* Bio-inspired computer vision: towards a synergistic approach of artificial and biological vision. **Computer Vision and Image Understanding**, [S.l.], v.150, p.1 – 30, 2016.

MEIER, L. *et al.* PIXHAWK: a system for autonomous flight using onboard computer vision. *In*: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 2011, Shanghai, China. **Proceedings [...]** IEEE, 2011. p.2992–2997.

MILFORD, M. J.; WYETH, G. F. SeqSLAM: Visual route-based navigation for sunny summer days and stormy winter nights. *In*: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 2012, Saint Paul, MN, USA. **Proceedings [...]** IEEE, 2012. p.1643–1649.

MONCHIERO, M. *et al.* Exploration of distributed shared memory architectures for NoC-based multiprocessors. **J. Syst. Archit.**, [S.l.], v.53, n.10, p.719–732, Oct. 2007.

MONTIEL, J.; CIVERA, J.; DAVISON, A. Unified inverse depth parametrization for monocular SLAM. *In*: ROBOTICS: SCIENCE AND SYSTEMS, 2006, Philadelphia, USA. **Proceedings [...]** IEEE, 2006.

MORENO, E. I. *et al.* Integrating abstract NoC models within MPSoC design. *In*: IEEE/IFIP INTERNATIONAL SYMPOSIUM ON RAPID SYSTEM PROTOTYPING, 19., 2008, Monterey, CA, USA. **Proceedings [...]** IEEE, 2008. p.65–71.

MOULON, P.; MONASSE, P.; MARLET, R. Global fusion of relative motions for robust, accurate and scalable structure from motion. *In*: IEEE INTERNATIONAL CONFERENCE ON COMPUTER VISION, 2013, Sydney, NSW, Australia. **Proceedings [...]** IEEE, 2013. p.3248–3255.

NAWAF, M. M.; TRÉMEAU, A. Monocular 3D structure estimation for urban scenes. *In*: IEEE INTERNATIONAL CONFERENCE ON IMAGE PROCESSING (ICIP), 2014, Paris, France. **Proceedings [...]** IEEE, 2014. p.3763–3767.

NOURANI-VATANI, N.; BORGES, P. V. K.; ROBERTS, J. M. A Study of feature extraction algorithms for optical flow tracking. *In*: AUSTRALASIAN CONFERENCE ON ROBOTICS AND AUTOMATION, 2012, Wellington, New Zealand. **Proceedings [...]** Australian Robotics and Automation Association, 2012.

PAPAZOGLOU, M. P. *et al.* Service-oriented computing: State of the art and research challenges. **Computer**, [S.l.], v.40, n.11, p.38–45, Nov. 2007.

PATTERSON, D. 50 Years of computer architecture: from the mainframe CPU to the domain-specific tpu and the open RISC-V instruction set. *In*: IEEE INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE - (ISSCC), 2018, San Francisco, CA, USA. **Proceedings [...]** IEEE, 2018. p.27–31.

PEREIRA, F. I. **High precision monocular visual odometry**. 2018. Tese (Doutorado em Engenharia Elétrica) — Universidade Federal do Rio Grande do Sul, Porto Alegre, 2018.

PEREIRA, F. I. *et al.* Monocular visual odometry with cyclic estimation. *In*: SIBGRAPI CONFERENCE ON GRAPHICS, PATTERNS AND IMAGES (SIBGRAPI), 30., 2017, Niteroi, Brazil. **Proceedings [...]** IEEE, 2017. p.1–6.

PEREIRA, F. I. *et al.* A novel resection-intersection algorithm with fast triangulation applied to monocular visual odometry. **IEEE Transactions on Intelligent Transportation Systems**, [S.l.], p.1–10, 2018.

QUAN, L.; LAN, Z. Linear N-point camera pose determination. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, [S.l.], v.21, n.8, p.774–780, Aug. 1999.

RAHIMI, A. *et al.* A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters. *In*: DESIGN, AUTOMATION TEST IN EUROPE, 2011, Grenoble, France. **Proceedings [...]** IEEE, 2011. p.1–6.

REINBRECHT, C. *et al.* Gossip NoC – Avoiding timing side-channel attacks through traffic management. *In*: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI (ISVLSI), 2016, Pittsburgh, PA, USA. **Proceedings [...]** IEEE, 2016. p.601–606.

ROSSI, D. *et al.* Energy efficient parallel computing on the PULP platform with support for OpenMP. *In*: IEEE CONVENTION OF ELECTRICAL ELECTRONICS ENGINEERS IN ISRAEL (IEEEI), 28., 2014, Eilat, Israel. **Proceedings [...]** IEEE, 2014. p.1–5.

ROSSI, D. *et al.* Multicore signal processing platform with heterogeneous configurable hardware accelerators. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, [S.l.], v.22, n.9, p.1990–2003, 2014.

SAVAS, S. **Utilizing heterogeneity in manycore architectures for streaming applications**. 2017. Dissertação (Mestrado em Engenharia Elétrica) — Halmstad University, 2017.

SCHIAVONE, P. D. *et al.* Quentin: an ultra-low-power PULPissimo SoC in 22nm FDX. *In*: IEEE SOI-3D-SUBTHRESHOLD MICROELECTRONICS TECHNOLOGY UNIFIED CONFERENCE (S3S), 2018, Burlingame, CA, USA, USA. **Proceedings [...]** IEEE, 2018. p.1–3.

SCHWAMBACH, V. *et al.* Application-level performance optimization: a computer vision case study on STHORM. **Procedia Computer Science**, [S.l.], v.29, p.1113–1122, Jan. 2014.

SHI, J.; TOMASI, C. Good features to track. *In*: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, 1994, Seattle, WA, USA. **Proceedings [...]** IEEE, 1994. p.593–600.

SHI, Y. **Reevaluating Amdahl's Law and Gustafson's Law**. 2013. Disponível em: https://www.researchgate.net/profile/Yuan_Shi12/publication/228367369_Reevaluating_Amdahl's_law_and_Gustafson's_law/links/562f9dd408ae8e1256876a0a.pdf. Acesso em: 03 maio 2019.

SIEVERS, G. *et al.* Evaluation of interconnect fabrics for an embedded MPSoC in 28 nm FD-SOI. *In*: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS (ISCAS), 2015, Lisbon, Portugal. **Proceedings [...]** [S.l.: s.n.], 2015. p.1925–1928.

SONG, S.; CHANDRAKER, M.; GUEST, C. C. High accuracy monocular SFM and scale correction for autonomous driving. **IEEE transactions on pattern analysis and machine intelligence**, [S.l.], v.38, n.4, p.730–743, 2016.

SPIKE, a RISC-V ISA Simulator. 2019. Disponível em: https://github.com/riscv/riscv-isa-sim. Acesso em: 09 mar. 2019.

SRINIVASAN, S. Extracting structure from optical flow using the fast error search technique. **International Journal of Computer Vision**, [S.l.], v.37, n.3, p.203–230, 2000.

STRASDAT, H.; MONTIEL, J. M. M.; DAVISON, A. J. Real-time monocular SLAM: Why filter? *In*: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND

AUTOMATION, 2010, Anchorage, AK, USA. **Proceedings [...]** IEEE, 2010. p.2657–2664.

THIES, W.; KARCZMAREK, M.; AMARASINGHE, S. StreamIt: a language for streaming applications. *In*: COMPILER CONSTRUCTION, 2002, Grenoble, France. **Proceedings [...]** Springer Berlin Heidelberg, 2002. p.179–196. (Lecture Notes in Computer Science).

TIAN, G.; HAMMAMI, O. Performance measurements of synchronization mechanisms on 16PE NOC based multi-core with dedicated synchronization and data NOC. *In*: IEEE INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS AND SYSTEMS - (ICECS 2009), 16., 2009, Yasmine Hammamet, Tunisia. **Proceedings [...]** IEEE, 2009. p.988–991.

TOMASI, C.; KANADE, T. Shape and motion from image streams under orthography: a factorization method. **International Journal of Computer Vision**, [S.l.], v.9, n.2, p.137–154, Nov. 1992.

TONG, J. G.; ANDERSON, I. D. L.; KHALID, M. A. S. Soft-core processors for embedded systems. *In*: INTERNATIONAL CONFERENCE ON MICROELECTRONICS, 2006, Dhahran, Saudi Arabia. **Proceedings [...]** IEEE, 2006. p.170–173.

TOPCUOGLU, H.; HARIRI, S.; WU, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.13, n.3, p.260–274, Mar. 2002.

TRABER, A.; GAUTSCHI, M.; SCHIAVONE, P. D. **RI5CY Core**: Datasheet. [S.l.]: University of Bologna, 2016. Disponível em: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf. Acesso em: 31 dez. 2018.

TRABER, A. *et al.* PULPino: a small single-core RISC-V SoC. *In*: RISCV WORKSHOP, 3., 2016, Redwood Shores, CA, USA. **Proceedings [...]** [S.l.: s.n.], 2016.

TRIGGS, B. Factorization methods for projective structure and motion. *In*: CVPR IEEE COMPUTER SOCIETY CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, 1996. **Proceedings [...]** IEEE, 1996. p.845–851.

TYLECEK, R. *et al.* The second workshop on 3D reconstruction meets semantics: challenge results discussion. *In*: ECCV 2018 WORKSHOPS, 2019, Cham. **Proceedings [...]** Springer, 2019. p.631–644.

VEDALDI, A.; GUIDI, G.; SOATTO, S. Moving forward in structure from motion. *In*: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, 2007, Minneapolis, MN, USA. **Proceedings [...]** IEEE, 2007. p.1–7.

WANG, C. *et al.* Service-oriented architecture on FPGA-based MPSoC. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.28, n.10, p.2993–3006, Oct. 2017.

WANG, T.-H.; CHANG, J.-Y.; CHEN, L.-G. Algorithm and architecture for object tracking using particle filter. *In*: IEEE INTERNATIONAL CONFERENCE ON MULTIMEDIA AND EXPO, 2009, New York, NY, USA. **Proceedings [...]** [S.l.: s.n.], 2009. p.1374–1377.

WATERMAN, A. *et al.* **The RISC-V Instruction Set Manual, Volume I**: Base User-Level ISA. [S.l.]: EECS Department, University of California, Berkeley, 2011.

WESTOBY, M. J. *et al.* 'Structure-from-Motion' photogrammetry: a low-cost, effective tool for geoscience applications. **Geomorphology**, [S.l.], v.179, p.300–314, Dec. 2012.

WOLF, W.; JERRAYA, A. A.; MARTIN, G. Multiprocessor system-on-chip (MPSoC) technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.27, n.10, p.1701–1713, Oct. 2008.

WU, C. Towards linear-time incremental structure from motion. *In*: INTERNATIONAL CONFERENCE ON 3D VISION, 2013, Seattle, WA, USA. **Proceedings [...]** IEEE, 2013. p.127–134.

WÄCHTER, E. W. **Integração de novos processadores em arquiteturas MPSOC**: um estudo de caso. 2011. Dissertação (Mestrado em Engenharia Elétrica) — PUCRS, Porto Alegre, 2011.

WÄCHTER, E. W.; BIAZI, A.; MORAES, F. G. HeMPS-S: a homogeneous NoC-based MPSoCs framework prototyped in FPGAs. *In*: INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP (ReCoSoC), 6., 2011, Montpellier, France. **Proceedings [...]** Piscataway, 2011. p.1–8.

WÖHLER, C. Triangulation-based approaches to three-dimensional scene reconstruction. *In*: WÖHLER, C. (ed.). **3D Computer Vision**: efficient methods and applications. London: Springer London, 2013. p.3–87. (X.media.publishing).

YANG, C.; OE, S.; TERADA, K. Estimation of three-dimensional motion information from optical flow using subspace method. **The Transactions of the Institute of Electrical Engineers of Japan. C**, [S.l.], v.121, n.7, p.1187–1194, July 2001.

ZHANG, L.; JI, Q. Image segmentation with a unified graphical model. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, [S.l.], v.32, n.8, p.1406–1425, Aug. 2010.

ZHAO, P. *et al.* A directional-edge-based real-time object tracking system employing multiple candidate-location generation. **IEEE Transactions on Circuits and Systems for Video Technology**, [S.l.], v.23, n.3, p.503–517, Mar. 2013.

ZIAKAS, D. *et al.* Intel® QuickPath Interconnect architectural features supporting scalable system architectures. *In*: IEEE SYMPOSIUM ON HIGH PERFORMANCE INTERCONNECTS, 18., 2010, Mountain View, CA, USA. **Proceedings [...]** IEEE, 2010. p.1–6.

ÖZYEŞIL, O. *et al.* A survey of structure from motion. **Acta Numerica**, [S.l.], v.26, p.305–364, May 2017.