UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Processamento de Rede Neocognitron para Reconhecimento Facial de Ambiente de Alto Desempenho GPU

GUSTAVO POLI LAMEIRÃO DA SILVA

São Carlos – SP
Agosto/2007

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

S586pr

Silva, Gustavo Poli Lameirão da.
Processamento da rede neocognitron para reconhecimento facial em ambiente de alto desempenho GPU / Gustavo Poli Lameirão da Silva. -- São Carlos : UFSCar, 2008.
108 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2007.

1. Arquitetura de computador. 2. Alto desempenho. 3. Rede neural neocognitron. 4. GPU. I. Título.

CDD: 004.22 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Processamento da Rede Neocognitron para Reconhecimento Facial em Ambiente de Alto Desempenho GPU"

GUSTAVO POLI LAMEIRÃO DA SILVA

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. José Hiroki Saito
(DC/UFSCar)

Prof. Dr. Hélio Crestana Guardia
(DC/UFSCar)

Prof. Dr. Ronaldo Augusto de Lara Gonçalves
(UEM/Maringá)

São Carlos
Agosto/2007

Dedico este trabalho de dissertação aos meus pais,
cujo ato de amor me conseberam e por amor me apoiam
e me inspiram a superar as adversidades da vida,
a Giuliana pelo amor, paciência e companherismo
a minha amada filha Luísa, que mesmo na distância
sempre é e será minha fonte de inspiração
e ao Prof. Saito pela coragem em ser meu orientador.

Agradecimentos

Dedico meus sinceros agradecimentos para:

– ao Professor Saito, pelo exemplo profissional, humildade, competência e dedicação. Mas principalmente por sua disponibilidade mesmos aos finais de semana e em horários não convensionais, em que sempre se encontrava disposto a ajudar, não apenas a mim, mas a todos os seus ”Filhos”. Mais que um orientador O Orientador!

– ao Alexandre Levada, grande e verdadeiro amigo e exemplo de humildade e competência profissional, sempre disposto a ajudar e a explicar os mistérios do universo da matemática, cuja revisão de reconhecimento de padrões, foi de suma importância para que eu pudesse concluir meu mestrado;

– ao senhor Sebastiano e a dona Cleusa, pelos finais de semana maravilhosos e tranquilos em sua chacara, descanso tão necessário ao corpo e a mente;

– a Jasmine, Moha, Gaia, Duda e Elis, pelas brincadeiras, companherismo, carinho e verdadeira amizade;

– ao Professor Mascarenhas pelas fantásticas aulas de Processamento de Imagem e Reconhecimento de padrões;

– ao Luis Gustavo Cantanheira, pelas conversas loucas e idéias insanas, mas que me ajudaram e escolher minha linha de pesquisa;

– ao John e ao Zorzan, amigos de laboratório, pelas divertidas ações de fazer um ”lab feliz”;

– ao Mairum, ”irmão” mais velho, pela ajuda com a FPGA e por ser o mais ativo na coleta de frutas que faziamos no final do ano aqui na UFSCar;

Resumo

Neste trabalho é apresentada a implementação da Rede Neural Neocognitron, usando uma arquitetura de computação de alto desempenho baseada em GPU (*Graphics Processing Unit*). O Neocognitron é uma rede neural artificial, proposta por Fukushima e colaboradores, constituída de vários estágios de camadas de neurônios, organizados em matrizes bidimensionais denominadas planos celulares. Para o processamento de alto desempenho da aplicação de reconhecimento facial usando neocognitron foi utilizado o CUDA (*Compute Unified Device Architecture*) como API (*Application Programming Interface*) entre o CPU e o GPU, da GeForce 8800 GTX da empresa NVIDIA, com 128 ALU's. Como repositórios de imagens faciais foram utilizados imagens faciais desenvolvido na UFSCar e o banco da Universidade de Carnegie Melon, CMU-PIE. O balanceamento de carga na arquitetura de processamento paralelo foi obtido considerando o processamento de uma conexão de neurônio como um *thread*, e um conjunto de *threads*, como um bloco, segundo a filosofia de desenvolvimento dentro deste ambiente. Os resultados mostraram a viabilidade do uso deste tipo de dispositivo como ferramenta de processamento de dados maciçamente paralelo e que quanto menor a granularidade da paralelização e a independência dos processamentos, melhor é seu desempenho.

Abstract

This work presents an implementation of the Neocognitron Neural Network, using a high performance computing architecture based on GPU (Graphics Processing Unit). Neocognitron is an artificial neural network, proposed by Fukushima and collaborators, constituted of several hierarchical stages of neuron layers, organized in two-dimensional matrices called cellular plains. For the high performance computation of Face Recognition application using Neocognitron it was used CUDA (Compute Unified Device Architecture) as API (Application Programming Interface) between the CPU and the GPU, from GeForce 8800 GTX of NVIDIA company, with 128 ALU's. As face image databases it was used a face database created at UFSCar, and the CMU-PIE (Carnegie Melon University - Pose, Illumination, and Expression) database. The load balancing through the parallel processing architecture was obtained by means of the distributed processing of the cellular connections as threads organized in blocks, following the CUDA philosophy of development. The results showed the viability of this type of device as a massively parallel data processing tool, and that smaller the granularity of the parallel processing, and the independence of the processing, better is its performance.

Tabela de Simbolos

Rede Neural Artificial Neocognitron

K_l	Número de planos de uma camada l
k_l	Plano da camada l
a_l	Peso-a
b_l	Peso-b
c_l	Peso-c
d_l	Peso-d
θ	Limiar de decisão (*threshold*)
S_l	Área de conexão
i	Posição dentro da Área de conexão
$u_{c_{l-1}}$	Posição da célula c da camada anterior $(l-1)$.
u_{S_l}	Valor da célula do plano-S

Programação Paralela

S_p	*Speed-up*
T_{seq}	Tempo de execução em um único processador
T_{par}	Tempo de execução utilizando uma máquina de p processadores
E_p	Eficiência (quanto o paralelismo foi explorado)
p	Número de processadores utilizados na paralelização

Programação Genérica com GPU

D_g	Especifica as dimensões (tamanho) do grid
D_b	Especifica as dimensões (tamanho) do bloco de threads
N_s	Especifica o Número de bytes da memória a ser compartilhada por block

Tabela de Siglas

CC-NUMA	NUMA com coerência de cache
CPU	Central Processing Unit
CUDA	*Compute Unified Device Architecture*
DLA	*Dynamic Link Architecture*
EGM	*Elastic Graph Matching*
FRT	*Face Recognition Techniques*
GPU	*Graphic Processor Unit*
HPC	*High Performance Computing*
LAN	*Local Network*
LDA	*Linear Discriminant Analysis*
MIMD	*Multiple Instruction Multiple Data*
MISD	*Multiple Instruction Single Data*
MPP	*Massive Parallel Processing*
NN	*Neural Networks*
NUMA	Acesso não-uniforme a Memória
PCA	*Principal Component Analysis*
RPC	*Remote procedure call*
SIMD	*Single Instruction Multiple Data*
SISD	*Single Instruction Single Data*
SMP	*Multiprocessador Simétrico*
UC	Unidade de Controle
PDA	*Personal Digital Assistent*

Sumário

Resumo **(i)**

Abstract **(ii)**

Tabela de Simbolos **(iii)**

Rede Neural Artificial Neocognitron . . . . . (iii)

Programação Paralela . . . . . (iii)

Programação Genérica com GPU . . . . . (iii)

Tabela de Siglas **(iv)**

Lista de Figuras **(x)**

Lista de Tabelas **(xiii)**

1 Introdução **1**

1.1 Motivação e Relevância 1

1.2 Objetivos e Definições do Problema 1

1.3 Estrutura da Dissertação 2

2 Reconhecimento de Faces **4**

2.1 Uma abordagem usando redes neurais 5

2.2 Conclusão 7

3 Rede Neural Artificial Neocognitron **8**

3.1 Sistema Visual dos Mamíferos 8

3.2 Rede Neocognitron 11
3.2.1 Arquitetura da Rede 11
3.2.2 Processamento da Rede 15
3.2.3 Treinamento da Rede 20
3.2.4 Conclusão 25

4 Computação de Alto Desempenho **26**
4.1 Arquiteturas Paralelas 26
4.1.1 SISD 26
4.1.2 SIMD 27
4.1.3 MISD 28
4.1.4 MIMD 29
4.2 Programação Paralela 30
4.2.1 Conceitos Básicos Sobre Programação Paralela 30
4.2.2 Tipos de Paralelismo Quanto ao Conteúdo 31
4.2.3 Tipos de Paralelismo Quanto a Forma 32
4.3 Exploração do Paralelismo 37
4.4 Ambiente Paralelo 37
4.4.1 Granularidade 38
4.4.2 Construção de Algoritmos Paralelos 38
4.5 Implementação de Programas Paralelos 39
4.5.1 Mapeamento 39
4.6 Avaliação do Desempenho 39
4.6.1 *Speed-up* 39
4.6.2 Eficiência 40
4.7 Ferramentas para Programação Paralela 40
4.7.1 Compiladores Paralelizantes 41

4.7.2 Extensões Paralelas 41
4.7.3 Linguagens Concorrentes 41
4.8 Conclusão 42

5 Programação Genérica com GPU **43**
5.1 Evolução do GPU 43
5.2 GPU como Dispositivo de Processamento Genérico 46
5.3 CUDA 50
5.3.1 Modelo de Programação 52
5.4 Conclusão 59

6 Trabalho Desenvolvido **61**
6.1 Trabalhos Anteriores 61
6.2 Processamento da Rede Neocognitron 62
6.3 Paralelização da Rede Neocognitron dentro do CUDA 68
6.4 Implementação do Projeto Enquanto Paradigma de HPC 72
6.4.1 HPC/GPU - Segundo seu Fluxo de Instruções de Dados 72
6.4.2 HPC/GPU - Tipo de Paralelismo Quanto ao Conteúdo 72
6.4.3 HPC/GPU - Tipo de Paralelismo Quanto à Forma 72
6.4.4 HPC/GPU - Granularidade 72
6.5 Algoritmo e Estrutura de Dados Implementado 73
6.6 Organização do Código 74
6.7 Estágios do Fluxo do Processamento da Aplicação 76
6.7.1 Rede Treinada 76
6.7.2 Carga de Pesos da Rede Treinada 76
6.7.3 Alocar Estrutura de Dados no GPU 77
6.7.4 Estrura de Dados no GPU 78

6.7.5 Imagem de Faces . . . 78
6.7.6 Carregar Imagem a ser Processada . . . 78
6.7.7 Processamento dos Estágios da Rede . . . 79
6.7.8 Executar Processamento Estágio X . . . 80
6.7.9 Transferir Resultados para Host . . . 80
6.7.10 Apresentar classe vencedora . . . 81
6.8 Ambiente de Desenvolvimento Utilizado . . . 81
6.9 Conclusão . . . 82

7 Resultados **83**
7.1 Acuracidade da Rede Neocognitron Processada no GPU/CUDA . . . 83
7.2 Tempos de Execução . . . 83
7.3 Desempenho do Ambiente GPU/CUDA . . . 85
7.4 Alocação dos Recursos do GPU para o Processamento da Rede . . . 86
7.4.1 Recursos Alocados para Processamento do Estágio 01 e 02 . . . 87
7.4.2 Recursos Alocados para Processamento do Estágio 03 . . . 88
7.5 Conclusões . . . 91

8 Conclusões e Trabalhos Futuros **92**
8.1 Conclusão do Trabalho . . . 92
8.2 Proposta para Trabalhos Futuros . . . 94

Referências Bibliográficas **96**

Apêndice A – Criando Makefile para Desenvolvimento de Projtetos CUDA em Linux **100**

Apêndice B – 1.1.Exemplo de Código Fonte para Início do Projeto **107**

Relação de Algoritmos

3.1 Algoritmo para computar um plano celular 22

3.2 Treinamento de um estágio da rede Neocognitron 24

Lista de Figuras

3.1 Sistema de Transmissão de Sinais dos Olhos ao Cérebro 9

3.2 Estrutura hierárquica das células do olho com o nervo óptico 10

3.3 Estágios da rede Neocognitron 12

3.4 Camadas da rede Neocognitron 12

3.5 Planos celulares da rede Neocognitron 13

3.6 Células da rede Neocognitron 14

3.7 Fluxo de pesos da rede Neocognitron 14

3.8 Os mesmos pesos agem em um mesmo plano 15

3.9 Exemplo da área de conexão de uma célula-V 16

3.10 Exemplo de área de conexão de uma célula-S 17

3.11 Processo de reconhecimento de padrão da célula-S 17

3.12 Processo de ativação da célula-C 19

3.13 Utilização do peso-a 19

3.14 Utilização do peso-b 20

3.15 Utilização do peso-c 21

3.16 Utilização do peso-d 21

4.1 Diagrama de blocos de uma arquitetura SISD 26

4.2 Fluxo de instruções sendo executadas na arquitetura SISD 27

4.3 Diagrama de blocos de uma arquitetura SIMD 27

4.4 Exemplo de fluxo de instruções na arquitetura SIMD 28

4.5 Diagrama de blocos de uma arquitetura MISD 28

4.6 Diagrama de blocos de uma arquitetura MIMD 29

4.7 Fluxo de instruções sendo executadas na arquitetura MIMD 29
4.8 Diagrama de Paralelização 30
4.9 Paralelismo de dados 31
4.10 Paralelismo functional 32
4.11 Paralelismo de Objetos 32
4.12 Fases Paralelas 33
4.13 Divisão e Conquista 34
4.14 *Pipeline* 35
4.15 Mestre/Escravo 35
4.16 *Pool* de trabalho 36
5.1 Modelo esquemático de processamento paralelo de dados 43
5.2 Pipeline do desenvolvimento tradicional em GPU 46
5.3 Operações de ponto flutuante por segundo (CPU x GPU) 47
5.4 GPU Destina mais Transistores para o Processamento de Dados 47
5.5 *Pipeline* tradicional da placa gráfica 48
5.6 Pilha de Software CUDA 51
5.7 GPU Acessando a memoória para leitura 51
5.8 GPU Acessando a memória para escrita 51
5.9 Memória compartilhada entre as ALUs de um bloco 52
5.10 Endereço das threads dentro dos blocos.interno às grades (*grid* em inglês) 54
5.11 Modelo de Memória no CUDA 55
5.12 Modelo do Hardware - Processamento SIMD de Dados 56
6.1 Diagrama de Blocos do Sistema de Reconhecimento Facial da rede Neocognitron dentro do projeto 62
6.2 Repositório de Faces desenvolvido na UFSCar 62
6.3 Repositório de Fases CMU-PIE 63

6.4 Amostra de uma imagem da base de dados CMU PIE, e obtenção de uma imagem facial usando captura da face e redução 63

6.5 Processamento da Rede Neocognitron 64

6.6 Representação gráfica da matriz de peso-c do estágio 1 65

6.7 Representação gráfica da matriz de peso-c do estágio 2 66

6.8 Representação gráfica da matriz de peso-c do estágio 3 66

6.9 Representação gráfica da matriz de peso-d do estágio 1 67

6.10 Representação gráfica da matriz de peso-d do estágio 2 68

6.11 Representação gráfica da matriz de peso-d do estágio 3 68

6.12 Relação entre a arquitetura CUDA e a rede Neocognitron 69

6.13 Divisão de ambiente quanto o processamento/desenvolvimento de funções 70

6.14 Diagrama do processamento das conexões do Neurônio 71

6.15 Estrutura de dados do estágio A 73

6.16 Estrutura de Dados alocada dentro do dispositivo 75

6.17 Diagrama de blocos da aplicação 75

6.18 Foto da estação de trabalho usada, com destaque a presença do GPU GeForce 8800GTX 82

7.1 Variação do Tamanho do Bloco para chamada ao kernel referente ao cálculo dos estágios 1 e 2 88

7.2 Variação da quantidade de registradores para chamada ao kernel referente ao cálculo dos estágios 1 e 2 88

7.3 Variação da memória compartilhada utilizada para chamada ao kernel referente ao cálculo dos estágios 1 e 2 89

7.4 Variação do Tamanho do Bloco para chamada ao kernel referente ao cálculo do estágio 3 90

7.5 Variação da quantidade de registradores para chamada ao kernel referente ao cálculo do estagio 3 90

7.6 Variação da memória compartilhada utilizada para chamada ao kernel referente ao cálculo do estágio 3 91

Lista de Tabelas

5.1 Comparação entre HPC x GPU 49

5.2 Formação do endereço da thread dentro do bloco 53

5.3 Formação do endereço do blockID dentro de um gride 53

5.4 Politicas de acesso a memória da GPU 55

5.5 Qualificadores de funções CUDA 58

5.6 Qualificadores de variáveis CUDA 59

5.7 Variáveis embutidas CUDA 59

6.1 Dimensionalidade dos pesos utilizados na rede 64

6.2 Matriz de pesos-c do estágio 1 65

6.3 Matriz de Peso-c do Estágio 2 65

6.4 Matriz de peso-c do Estágio-3 66

6.5 Matriz de peso-d do Estágio 1 67

6.6 Matriz de peso-d do Estágio 2 67

6.7 Matriz de peso-d do Estágio 3 67

6.8 Correspôndencias entre as arquiteturas CUDA x Neocognitron 70

6.9 Modelo de dimensionalidade por estágio da rede a ser executado 71

6.10 Relação das Ferramentas utilizadas no projeto 81

7.1 Taxa de acudarcidade do processamento da rede Neocognitron GPU/CUDA 83

7.2 Tempo de execução por estágio da rede 84

7.3 Comparação de Tempo de Processamento Entre Arquiteturas 86

7.4 Speed-up e Eficiência do SIMD x MIMD 86

7.5 Demonstrativo de áreas alocadas 86

7.6 Tabela de recursos usados na chamada ao kernel para calculo do estagio 1 e 2 da rede 87

7.7 Tabela de recursos usados na chamada ao kernel para calculo do estagio 3 da rede . 89

1 Introdução

1.1 Motivação e Relevância

O reconhecimento de faces por máquinas é uma área de pesquisa ativa e emergente, envolvendo várias disciplinas, como processamento de imagens, reconhecimento de padrões, visão computacional, arquitetura de computadores e redes neurais. Existem numerosas aplicações comerciais de Técnicas de Reconhecimentos Faciais (FRT) tais como verificação de face para controle de acesso e aplicações de segurança/vigilância usando câmeras de vídeo.

Diversos países já utilizam esta técnica para as mais diversas finalidades. Na China, por exemplo, foi desenvolvido um sistema para a identificação de imigrantes, sendo utilizado nas cidades de Shenzhen e Zhuhai (NOTíCIAS, 2006), apresentando ótimos resultados principalmente quando a impressão digital não pode ser mais analisada ou pela idade, por trabalho ou pela reação de produtos químicos. No mercado, empresas como a japonesa Omron, já possuem tecnologias proprietárias (Okao Vision) para serem utilizadas em sistemas de PDA's e telefones celulares (IDG, 2006).

Devido à natureza amigável ao usuário (não intrusiva), o reconhecimento de faces é atraente apesar da existência de métodos extremamente confiáveis de identificação pessoal biométrica tais como análise de impressões digitais e varredura da íris.

1.2 Objetivos e Definições do Problema

Como se pode perceber existem grandes desafios quanto às questões de reconhecimento facial, onde se pode destacar uma relação entre duas variáveis básicas do processo: o grau de confiabilidade/robustez da técnica a ser empregada e o custo computacional desta técnica.

O objetivo desta dissertação de mestrado é o estudo de uma arquitetura de reconhecimento facial que vise o aumento do desempenho por meio do uso de processamento maciço de dados, paralelizando o processamento de uma rede neural artificial Neocognitron em uma arquitetura de

computação de alto desempenho baseada no GPU.

Para se ter acesso o GPU como um dispositivo de programação de fins genéricos utiliza-se o CUDA, uma biblioteca que estende as funções da linguagem C de forma a disponibilizar o GPU como um dispositivo de processamento de dados (NVIDIA..., 2007).

1.3 Estrutura da Dissertação

Este documento encontra-se organizado dentro dos seguintes capítulos:

Capítulo 1 - Introdução: este presente capítulo, onde é realizada uma breve introdução sobre o cenário de onde nascem os objetivos e necessidades do trabalho além da descrição da organização deste documento.

Capítulo 2 - Reconhecimento de Faces: neste capítulo é apresentada uma descrição do cenário de reconhecimento facial, técnicas e principais necessidades.

Capítulo 3 - Rede Neural Artificial Neocognitron: neste capítulo é realizada uma descrição da rede neural artificial Neocognitron, que tem sua inspiração no sistema de visão dos mamíferos, concentrando-se na base conceitual proposta por Fukushima e colaboradores.

Capítulo 4 - Computação de Alto Desempenho: neste capítulo é apresentada a estrutura e filosofia que envolve o cenário da computação de alto desempenho (HPC) bem como os paradigmas de programação paralela.

Capítulo 5 - Programação Genérica com GPU: faz-se aqui uma descrição sobre o GPU, apresentando sua evolução histórica, passando de um dispositivo de processamento gráfico e controle de saída de vídeo até um dispositivo de processamento maciço de dados. Também se encontra aqui uma descrição detalhada sobre a arquitetura CUDA, uma proposta da empresa nVidia para processamento de dados de fins gerais dentro do GPU.

Capítulo 6 - Trabalho Desenvolvido: faz-se a descrição da implementação do projeto para o uso do GPU, com o uso do CUDA como arquitetura de processamento de dados para o reconhecimento de facial usando a rede Neocognitron como ferramenta de reconhecimento/processamento de dados.

Capítulo 7 - Resultados: faz-se aqui a apresentação dos resultados obtidos pela implementação do projeto proposto no capítulo 6.

Capítulo 8 - Conclusões e Trabalhos Futuros: aqui se encontram as conclusões a respeito dos resultados obtidos pelo uso da plataforma implementada e uma relação de propostas para

trabalhos futuros com o objetivo de dar continuidade na pesquisa apresentada nesta dissertação de mestrado.

2 *Reconhecimento de Faces*

O reconhecimento facial é desafiante e, até o momento, não existe nenhuma técnica que forneça uma solução robusta para todas as situações e diferentes aplicações em que pode ser empregado. Num contexto geral, o reconhecimento facial em cenários complexos não teria solução para os próximos anos. Contudo, existe uma esperança para contextos e aplicações específicos se algumas novas técnicas forem desenvolvidas e combinadas.

O reconhecimento facial pode ser dividido em duas aplicações básicas: identificação e verificação. No problema da identificação, a face que deve ser reconhecida não é conhecida e é comparada (*matched*) com imagens faciais de uma base de dados de imagens de indivíduos conhecidos. No problema da verificação, o sistema confirma ou rejeita a identidade proposta de uma imagem facial.

Embora possam existir diferenças, a descrição seguinte trata do problema geral de reconhecimento facial, sem uma distinção entre os dois problemas, uma vez que os desafios e as técnicas usadas são basicamente os mesmos.

As abordagens sobre reconhecimento facial para imagens estáticas podem ser agrupadas em técnicas de comparação (*matching*) geométricas e de padrões. No primeiro caso, as características geométricas das faces, tais como distâncias entre os fatores faciais, são comparadas. Essa técnica provê resultados limitados embora tenha sido usada extensivamente no passado. No segundo caso, imagens faciais representadas como uma matriz bi-dimensional de valores de intensidades de pixels é comparada com um único ou vários padrões que representam toda a face.

A abordagem mais bem sucedida de comparação de padrões usa a análise de componentes principais (PCA) ou análise de discriminantes lineares (LDA) para realizar a redução de dimensionalidade obtendo bom desempenho num tempo e complexidade computacional razoável. Outros métodos de comparações de padrões usam classificação por redes neurais e padrões deformáveis, tais como comparação com grafo elástico (EGM).

Recentemente, um conjunto de abordagens que usam diferentes técnicas para correção de distorções de perspectiva tem sido proposto. Essas técnicas são referidas como tolerantes ao ponto

de vista (*view-tolerant*). As referências de Zhao, Chellapa e outros (ZHAO et al., 2002) (CHELLAPPA; WILSON; SIROHEY, 1995) fornecem uma visão sobre os diversos tópicos de reconhecimento de faces. Os métodos de comparação de padrões são relatados por Moghddam, Turk, Sewets, Belhumeur, Zhao e outros (MOGHDDAM; PENTLAND, 1997) (TURK; PENTLAND, 1991) (SEWETS; WENG, 1996) (BELHUMEUR; HESPANHA; KRIEGMAN, 1997) (ETEMAD; CHELLAPPA, 1997) (ZHAO; CHELLAPA; KRISHNASWAMY, 1998). Os esquemas baseados em comparação de fatores geométricos são descritos por Manjunath, Okada e outros (MANJUNATH; CHELLAPPA; MALSBURG, 1992) (OKADA et al., 1998) e métodos híbridos são descritos por Pentland e outros (PENTLAND; MOGHADDAM; STARNER, 1994).

Embora todos esses tipos de sistemas tenham sido usados com sucesso para o reconhecimento facial, todos apresentam vantagens e desvantagens. Assim, esquemas apropriados devem ser escolhidos baseado nos requisitos específicos de uma dada tarefa. Por exemplo, sistemas baseados em EGM (OKADA et al., 1998) têm bom desempenho em geral, contudo, exigem um tamanho grande de imagem, por ex., 128x128. Isso restringe severamente a sua aplicação para vigilância baseada em vídeo, onde o tamanho da imagem facial é muito pequeno. Por outro lado, o sistema LDA de subespaço (ZHAO; CHELLAPPA; PHILLIPS, 1999) funciona bem com imagens grandes e pequenas, por exemplo, 96x84 ou 24x21.

2.1 Uma abordagem usando redes neurais

Redes neurais (NN) têm sido usadas no reconhecimento facial para diversos problemas: classificação de gênero (sexo), reconhecimento de faces, classificação de expressões faciais. Uma das demonstrações iniciais de NN em reconhecimento de faces foi o uso de mapa associado de Kohonen (KOHONEN, 1988). Usando um pequeno conjunto de imagens faciais, uma precisa revocação foi reportada mesmo quando a imagem de entrada era muito ruidosa ou com falta de partes. Essa capacidade tem também sido demonstrada usando hardware óptico (ABU-MOSTAFA; PSALTIS, 1987).

Brunelli e Poggio (BRUNELLI; POGGIO, 1992) descrevem uma abordagem de NN para a classificação de gênero usando um vetor de 16 atributos numéricos tais como espessura da sobrancelha, profundidade do nariz e lábios, seis raios do queixo, etc. Duas redes HyperBF (POGGIO; GIROSI, 1990) foram treinadas, uma para cada gênero. As imagens de entrada foram normalizadas com respeito aă variação de escala e rotação usando as posições dos olhos, que foram detectados automaticamente. O vetor de fatores de dimensão 16 foi também automaticamente extraído. As saídas das duas redes HyperBF foram comparadas, com a rotulação do gênero para a imagem teste sendo determinada pela rede com maior valor de saída. Nos experimentos reais de classificação

somente um subconjunto do vetor de fatores foi usado.

Com uma base de dados consistindo de imagens faciais de 21 homens e 21 mulheres, a estratégia deixe-um-fora (FUKUNAGA, 1989) foi empregada para classificação. Quando o vetor de fatores do conjunto de treinamento foi usado como vetor de teste, 92,5% de reconhecimento correto foi reportado; para faces não pertencentes ao conjunto de treinamento, a acuidade caiu para 87,5%. Alguma validação dos resultados de classificação automática tem sido reportada.

Usando um vetor de fator expandido para 35 dimensões, e um HyperBF por pessoa, a abordagem de classificação de sexo foi estendido para o reconhecimento facial. A motivação para a estrutura delineada é o conceito do neurônio avó: um único neurônio (a função Gaussiana na rede HyperBF) para cada pessoa. Como existiram um número relativamente pequeno de imagens de treinamento por pessoa, uma base de dados sintética foi gerada perturbando a média dos vetores de fatores disponíveis, e esses vetores foram usados para amostras de teste. Foram obtidos os resultados de classificação, para diferentes conjuntos de parâmetros de sintonização (coeficientes, centros e métricas dos HyperBFs). Alguma confirmação do comportamento caricatural das redes HyperBF , por estudos psicofísicos, foi também apresentada.

Os sistemas apresentados por Buhmann, Ladesraj e outros (BUHMANN; LADES; MALSBURG, 1990) (LADESRAJ et al., 1993) foram baseados na arquitetura *Dynamic Link Architecture* (DLA). DLAs tentam resolver alguns dos problemas conceituais de redes neurais artificiais convencionais, sendo o problema mais proeminente a expressão do relacionamento sintático em redes neurais.

DLAs usam plasticidade sináptica e são capazes de formar imediatamente conjuntos de neurônios agrupados em grafos estruturados e manter a vantagem de sistemas neurais. Ambos, Buhmann e Ladesraj usam wavelets baseados em Gabor para os fatores. Um mínimo de dois níveis, o domínio da imagem e o domínio do modelo, são necessários para um DLA. O domínio da imagem corresponde aă área do córtex visual primário e o domínio do modelo ao córtex temporal da visão biológica. O mecanismo DLA é baseado num formato de dados capaz de codificar informações sobre atributos e links no domínio da imagem e transportar essa informação para o domínio do modelo sem incluir a posição do domínio da imagem.

Nessa arquitetura a estrutura do sinal é determinada por 3 fatores: a imagem de entrada, excitação aleatória e espontânea dos neurônios, e a interação com as células dos mesmos nós ou nós vizinhos no domínio da imagem. A ligação entre os neurônios é codificada na forma das correlações temporais e é induzida pelas conexões excitatórias dentro da imagem.

Quatro tipos de ligações são relevantes para o reconhecimento de objetos e representações: ligando juntos todos os neurônios e células que pertencem ao mesmo objeto, expressando a rela-

ção de vizinhança na imagem do objeto, juntando as células de fatores para fatores em diferentes localizações, e ligando pontos correspondentes no grafo da imagem e grafo do modelo entre si. O mecanismo básico do DLA, além do parâmetro básico de conexão entre pares de neurônios, é uma variável dinâmica (J) entre pares de neurônios (i, j). As variáveis J desempenham um papel de pesos sinápticos para transmissão de sinais. Os parâmetros de conexão meramente agem para restringir as variáveis J. Os pesos de conexão $J_{i,j}$ são controlados pelas correlações do sinal entre os neurônios i e j. As correlações de sinais negativos levam a um decrescimento, e correlações de sinais positivos levam ao aumento em $J_{i,j}$. Na ausência da correlação, $J_{i,j}$ lentamente retorna a um estado de repouso. Cada imagem armazenada é apresentada pelo posicionamento apropriado de uma grade retangular de pontos sobre a imagem e armazenando cada flecha determinada localmente por pontos de grade. O reconhecimento de uma nova imagem é realizado mapeando a imagem na grade de flechas e comparando com todas as imagens armazenadas. A conformação do DLA é feita estabelecendo e modificando dinamicamente as ligações entre os pontos de grade.

A arquitetura DLA tem sido estendido para EBGM (*Elastic Bunch Graph Matching*) (WISKOTT; FELLOUS; MALSBURG, 1997) (WISKOTT; FELLOUS; MALSBURG, 1997), que é similar ao método descrito, mas ao invés de considerar somente uma flecha em cada nó, um conjunto de flechas é considerado, cada uma derivada de uma diferente imagem facial. Para manipular o problema de variação de postura no reconhecimento facial, a postura da face é primeiro determinada usando informação a priori (KRUGER; POTZSCH; MALSBURG, 1997) e as transformações dos conjuntos sob variações de pose são aprendidos (MAURER; MALSBURG, 1996). Os sistemas baseados na abordagem EBGM tem sido aplicados em detecção facial, extração de fatores, estimação de posturas, classificação por sexos, reconhecimento de imagens baseados em *sketch*, e reconhecimento geral de objetos. ãL' sugerido que o sucesso do sistema DLA/EBGM pode ser devido Ãă semelhança com o sistema visual humano (BIEDERMAN; KALOCSAI, 1998).

2.2 Conclusão

O uso de redes neurais artificiais tem contribuído com a área de reconhecimento facial por ser uma ferramenta com a capacidade de aprendizagem sobre os padrões que devem reconhecer. Dentre os diversos tipos de redes neurais existentes a Neocognitron é uma das mais adequadas para esta tarefa, pelo fato de ser uma rede especialista que simula o funcionamento do sistema visual dos mamíferos. Mais detalhes sobre esta rede pode ser encontrado no próximo capítulo.

3 *Rede Neural Artificial Neocognitron*

O Neocognitron é um modelo de rede neural proposto por Fukuhsima (FUKUSHIMA, 1979) (FUKUSHIMA; MIYAKE, 1982) (FUKUSHIMA; WAKE, 1992) (FUKUSHIMA; TANIGAWA, 1996) (SAITO; FUKUSHIMA, 1998) (FUKUSHIMA, 2003) com um propósito prático específico: o reconhecimento de caracteres escritos aă mão. Ao propor o Neocognitron, Fukushima e seus colegas estavam interessados em desenvolver um modelo de sistema visual baseado nos estudos de visão biológica de Hubel e Wiesel (HUBEL; WIESEL, 1962) (HUBEL; WIESEL, 1968) do campo receptivo de células do córtex estriado de gato e macaco.

Os achados de Hubel e Wiesel mostraram que as células da área 17 do cérebro de um gato respondem preferencialmente ou exclusivamente para estímulos espaciais com certas estruturas. Em outras palavras, estas células tinham preferência para orientação espacial e pareciam responder melhor a linha clara ou escura de largura específica, assim como a bordas de larguras específicas em diversas orientações espaciais.

3.1 Sistema Visual dos Mamíferos

A via visual nos mamíferos pode ser dividida em sete ou oito estágios. Os três primeiros estágios estão concentrados na retina, sendo que no primeiro estágio temos as células nervosas especiais chamadas receptoras.

O nervo óptico, que carrega toda a saída da retina, é um feixe de axônios de células ganglionares que saem do olho em direção ao cérebro. O nervo óptico prossegue até o núcleo geniculado lateral, que por sua vez, envia sua saída ao córtex estriado, contendo três ou quatro estágios. As via de transmissão dos impulsos nervosos que deixam a retina como pode ser observado na Figura 3.1.

A retina é a parte neural do olho situada na parte posterior, sendo nela a ocorrência dos primeiros processamentos da imagem. A parte do olho ideal para estudos, por diversas razões; entre elas:

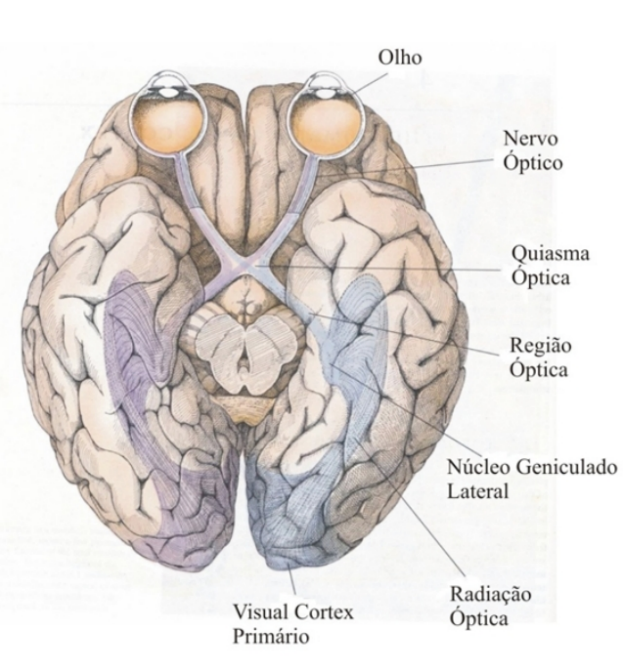

Figura 3.1: Sistema de Transmissão de Sinais dos Olhos ao Cérebro

- Facilidade de remoção, por estar na parte posterior do olho; possui uma organização anatômica bem definida e altamente organizada;
- Conter relativamente poucas classes básicas de células;
- Permitir em muitos vertebrados que apresentem células relativamente grandes, que os registros extracelulares sejam feitos rotineiramente;
- Em todas as espécies de vertebrados é encontrada uma organização anatômica semelhante e relativamente simples, e
- Facilidade de se estimular a retina naturalmente focalizando padrões de luz nos receptores.

Uma ampla variedade de respostas de células tem sido registrada em muitas espécies. Algumas dessas respostas indicam uma surpreendente complexidade de processamento de informações na retina.

As células fotossensíveis, que são os cones e bastonetes (receptores da visão), estabelecem sinapses com as células bipolares que, por sua vez, conectam-se com as células ganglionares. A luz incidente sobre a retina deve atravessar todas as camadas internas do olho para atingir os fotoreceptores. A excitação desses pela luz dá origem a impulsos nervosos que caminham em direção oposta aquela seguida pelo raio luminoso, ou seja das células fotossensíveis para as células bipolares e destas para as células ganglionares.

A camada do meio da retina (entre os cones e bastonetes e as células ganglionares) contém três tipos de células: bipolares, horizontais e amácrinas. As células bipolares recebem entradas dos receptores e muitas delas estão conectadas diretamente nas ganglionares. As células horizontais ligam-se aos receptores, às bipolares e às amácrinas, que por sua vez, ligam as bipolares às ganglionares.

Na camada de células na frente da retina estão as células ganglionares que são os neurônios para entrada da retina. Estas são as células mais estudadas, pois por serem planares podem ser tratadas como sendo bidimensionais, o que facilita, por exemplo, seu traçado.

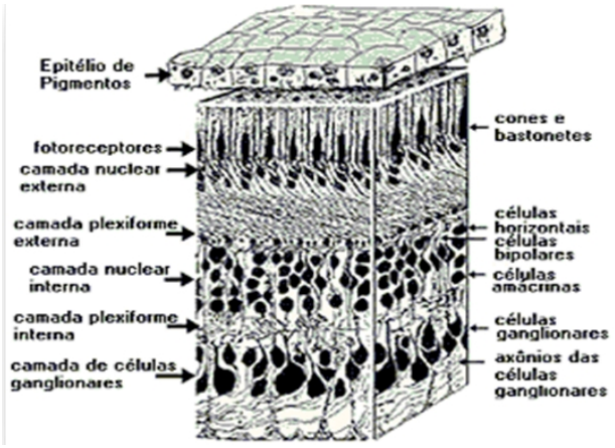

Figura 3.2: Estrutura hierárquica das células do olho com o nervo óptico

As retinas de várias espécies têm sido estudadas e algumas parecem ser mais simples que outras. As regiões de algumas retinas têm vantagens únicas, como é o caso, por exemplo, da região central da retina dos primatas, onde as células bipolares anãs e as ganglionares apresentam terminais de axônios e campos dendríticos limitados. Um arranjo torna possível traçar no microscópio eletrônico o processo pré e pós-sináptico dessas células.

O núcleo geniculado lateral é responsável por enviar as informações ao córtex visual. As fibras do nervo óptico fazem sinapses com as células no núcleo geniculado lateral e os axônios destas células terminam no córtex visual. As conexões dos olhos para o geniculado e desta para o córtex são topograficamente muito bem organizadas.

No córtex visual a imagem é apropriadamente interpretada. No estágio primário desse córtex os neurônios respondem a características simples, como segmentos de retas. Posteriormente os neurônios processam informações mais sofisticadas, como por exemplo cor, movimento, profundidade e até o reconhecimento de objetos específicos.

Estudos detalhados sobre a visão biológica foram realizados por Hubel e Wiesel (HUBEL; WIESEL, 1962) (HUBEL; WIESEL, 1968).

3.2 Rede Neocognitron

A rede Neocognitron tem como princípio básico de funcionamento a extração de características de forma hierárquica, ou seja, realiza a extração das características em vários estágios. No primeiro estágio, as características extraídas são as mais simples, resumindo-se a linhas em diferentes sentidos de rotação, e nos estágios seguintes as características vão se apresentando mais complexas. Uma característica desta rede é que os fatores extraídos por um estágio têm como informação de entrada apenas as enviadas pelo estágio anterior.

3.2.1 Arquitetura da Rede

Os estágios de uma rede Neocognitron estão organizados em camadas, sendo que cada uma dessas camadas possui seu próprio tipo/complexidade de dados a ser processado, sendo estas constituídas de células simples (células-S), células complexas (células-C) e células de atividade (células-V).

Dentro de cada camada existe um conjunto de planos celulares, onde ficam organizadas as células, cada qual especializada dentro do tipo/complexidade da camada em que esta se encontra inserido. A exceção aqui é a camada de atividade que possui apenas um único plano celular com o mesmo tamanho dos planos celulares da camada de células simples de seu estágio.

Estágios

Os estágios de uma rede Neocognitron funcionam como uma ferramenta de organização do processo de extração das características ou fatores, onde em cada estágio existe certo grau de complexidade da característica extraída do padrão. O primeiro estágio, chamado de estágio zero (Estágio 0), não é usado dentro do esquema hierárquico de extração de características, sendo usado como a retina do olho, que apresenta o padrão a ser processado pela rede. A Figura 3.3 apresenta os estágios de uma rede Neocognitron com cinco estágios.

O número de estágios de uma rede Neocognitron depende do tamanho do padrão de entrada a ser processado pela rede. Quanto maior o tamanho do padrão de entrada maior o número de estágios necessários para a rede. Por exemplo, um padrão de entrada de 20 x 20 pixels, resulta tipicamente em uma rede de três estágios hierárquicos.

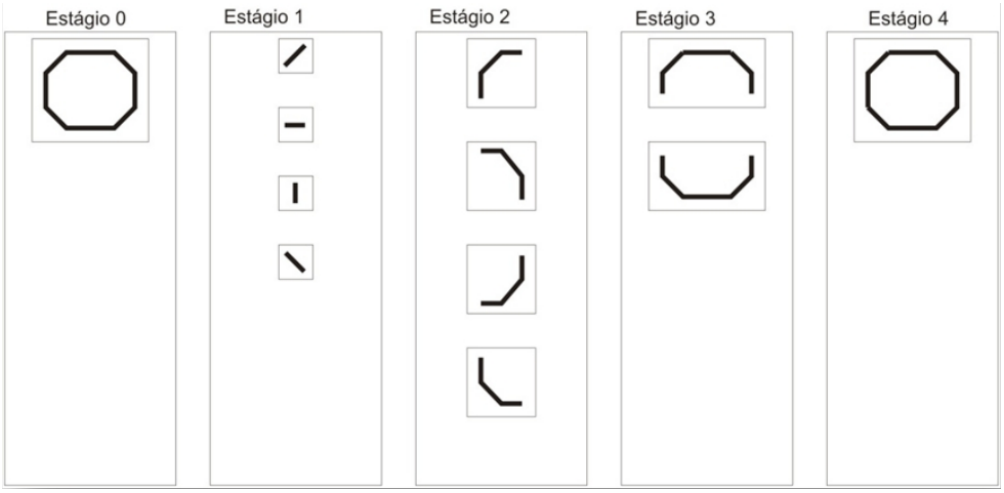

Figura 3.3: Estágios da rede Neocognitron

Camadas

Cada estágio de uma Rede Neocognitron é dividida em três camadas: uma camada simples (camada-S), uma camada complexa (camada-C) e uma camada de atividade (camada-V). Assumindo a rede Neocognitron com cinco estágios, apresentada na seção anterior (Figura 3.3), a sua representação em camadas pode ser observada na Figura 3.4.

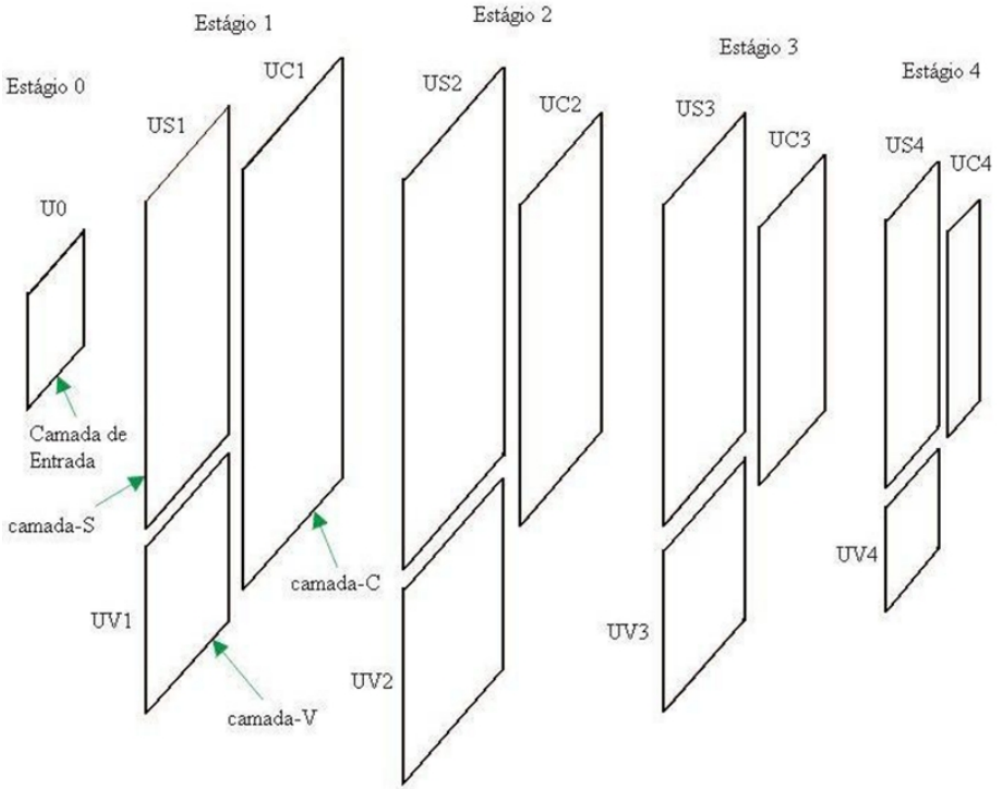

Figura 3.4: Camadas da rede Neocognitron

No estágio 0 existe apenas uma única camada, que é a camada de entrada ou padrão de entrada.

Todos os demais estágios possuem três tipos de camadas, uma Camada-S, uma Camada-V e uma Camada-C.

Planos Celulares

Cada camada é formada por um certo número de planos celulares. O número de planos celulares em cada Camada-S e Camada-C está relacionado com o número de características extraídas pelo estágio da rede. Numa Camada-V existe apenas um único plano celular. O tamanho dos planos é igual para uma mesma camada e vai decrescendo à medida que sobe a hierarquia dos estágios. A Figura 3.5 mostra os planos celulares distribuídos nas camadas da rede.

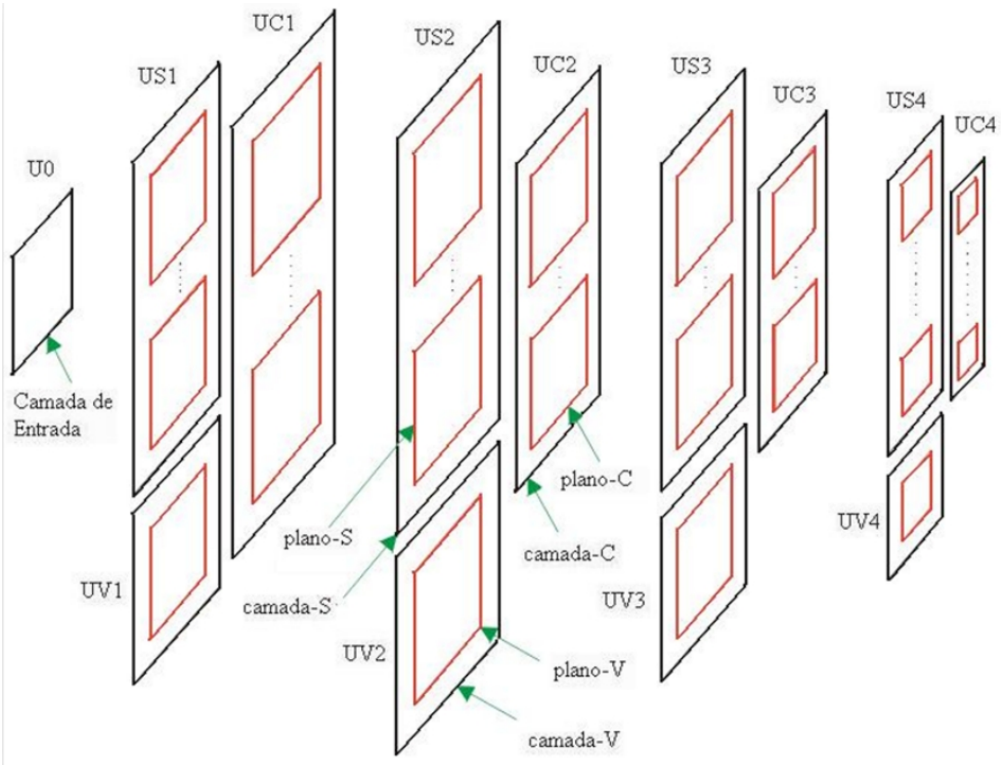

Figura 3.5: Planos celulares da rede Neocognitron

Células

A menor unidade da estrutura da rede Neocognitron são as células, organizadas, dentro dos planos celulares. Cada Plano-S, Plano-V, Plano-C e Camada de entrada são formadas por um conjunto (*array*) de células especializadas.

A Figura 3.6 mostra as células distribuídas ao longo dos planos da rede. Um plano-C da camada UC4 do último estágio da rede contém apenas uma única célula, cuja atividade indica o reconhecimento do padrão de entrada.

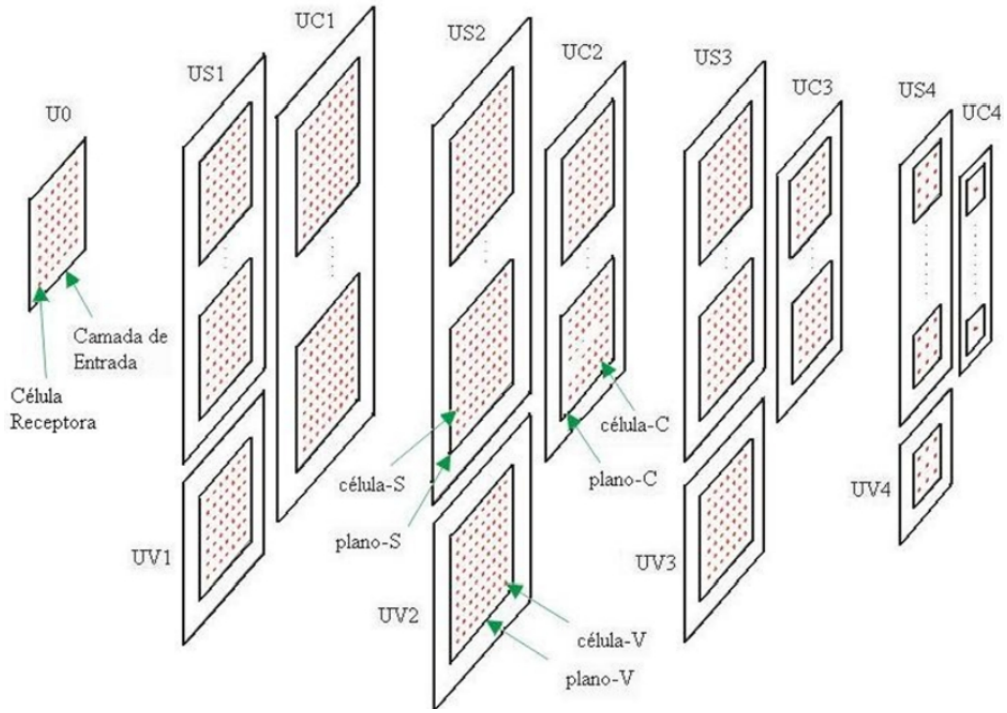

Figura 3.6: Células da rede Neocognitron

Pesos e Conexões

Uma característica da rede Neocognitron é ter um grande número de células mas um número reduzido de conexões. As células são conectadas a uma área de conexão reduzida, da camada anterior. Essa característica de conectividade é diferente dos Perceptrons Multicamadas, em que um neurônio de uma camada é conectado a todos os neurônios da camada anterior.

Para cada conexão existe um peso, que é utilizado para influenciar na quantidade de informação que é transferida. Existem na rede Neocognitron quatro tipos de pesos: peso-a, peso-b, peso-c e peso-d, cujas utilizações são resumidas conforme Figura 3.7.

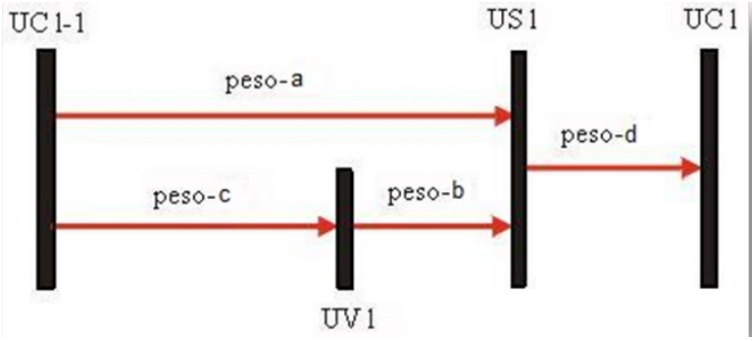

Figura 3.7: Fluxo de pesos da rede Neocognitron

Dentro de um plano celular, todas as suas células compartilham o mesmo peso; isso faz com que todas as células desse plano passem a observar a mesma característica, desta forma especiali-

zando o plano para a respectiva característica.

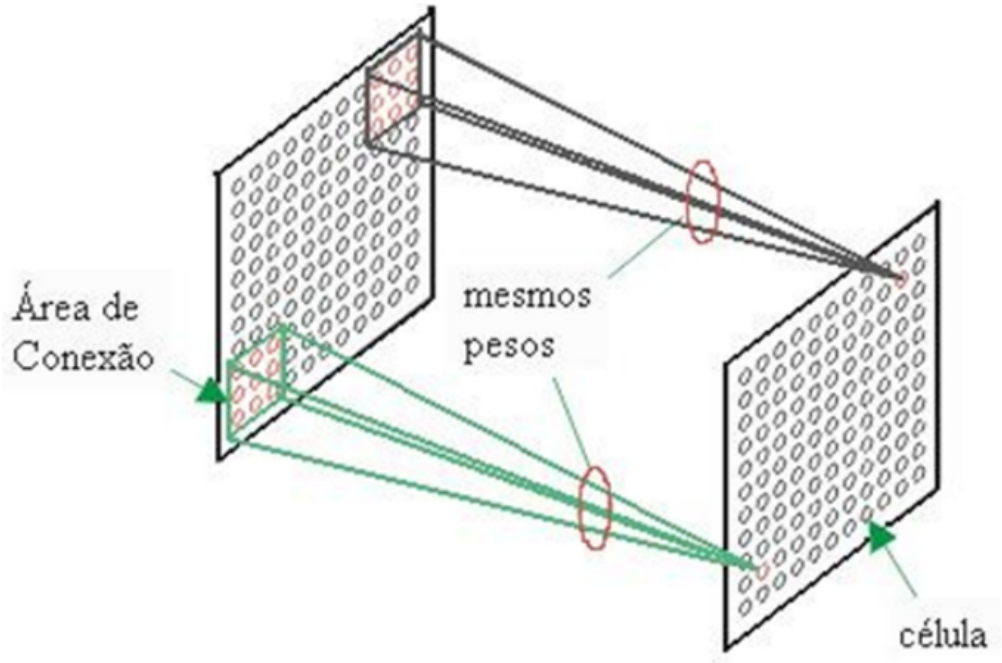

Figura 3.8: Os mesmos pesos agem em um mesmo plano

Pode-se ainda organizar os pesos em duas categorias, os que são modificados por treinamento (peso-a e peso-b) e os que não são modificados, ou seja, os valores que lhe são atribuídos continuam inalterados desde a implementação da rede (peso-c e peso-d).

3.2.2 Processamento da Rede

Célula-V

Cada Célula-V calcula a atividade dos valores das Células-C provenientes de uma área de conexão da Camada-C anterior. O tamanho da área de conexão é o mesmo para as Células-V e Células-S em um estágio da rede e é determinado no momento da construção da rede. Um exemplo da área de conexão pode ser observada na Figura 3.9.

O valor da Célula-V representa a média de atividades das células pertencentes àă sua área de conexão e é usado de forma a inibir as Células-S correspondentes. A exata especificação da função da Célula-V é dada pela Equação 3.1:

$$u_{vl}(n) = \sqrt{\sum_{k_{l-1}}^{K_{l-1}} \sum_{i \in S_l} c_l(i).u^2_{c_{l-1}}(n+1, k_{l-1})} \tag{3.1}$$

onde o peso-c deve ser ≥ 0.

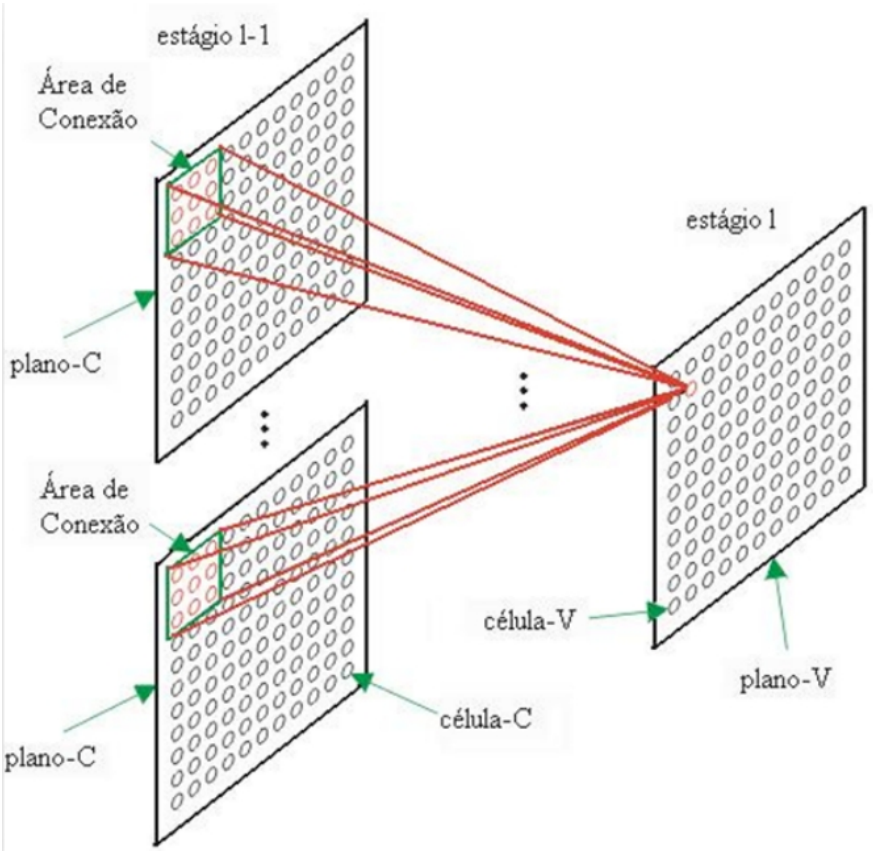

Figura 3.9: Exemplo da área de conexão de uma célula-V

Célula-S

A Célula-S avalia os valores de saída das Células-C de uma área de conexão da Camada-C do estágio anterior, ou camada de entrada. Conforme visto na seção anterior, o tamanho da área de conexão é o mesmo para as Células-S e Células-V de um mesmo estágio.

A função da Célula-S é reconhecer uma característica na área de conectada. Para reconhecer uma característica, uma Célula-S usa as informações existentes na área de conexão e as informações a respeito de atividades nesta área, informada pela Célula-V. A característica extraída por uma Célula-S é determinada por pesos nas suas conexões de entrada.

A extração de características por um plano-S e o significado dos pesos é mais fácil de ser observado nas células da camada u_{S_l} (primeira camada) da rede. Em cada Célula-S desta camada, existe apenas uma área de conexão e esta área é o campo receptivo ou área de conexão do padrão de entrada. Então, os pesos carregam consigo a representação de alguma característica.

Além disso, todas as células de um plano-S são iguais, ou seja, são portadoras dos mesmos pesos nas suas conexões de entrada. A Figura 3.11 mostra à esquerda a entrada U0 e à direita um plano US_1. Como todas as células são iguais, qualquer célula é capaz de reconhecer uma mesma

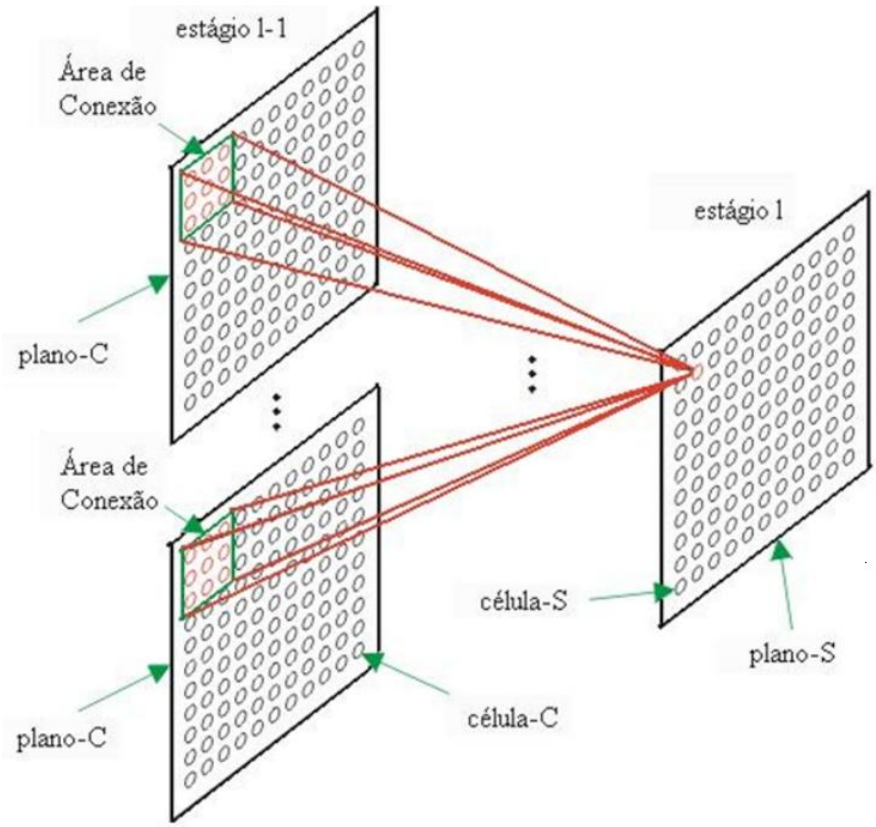

Figura 3.10: Exemplo de área de conexão de uma célula-S

característica embutida nos pesos que, no exemplo, é um traço vertical porém em posicionamentos diferentes. Assim, aquela célula-S que está posicionada na área de conexão que contem a característica (traço vertical) responde, conforme assinalado no plano-S da Figura 3.11.

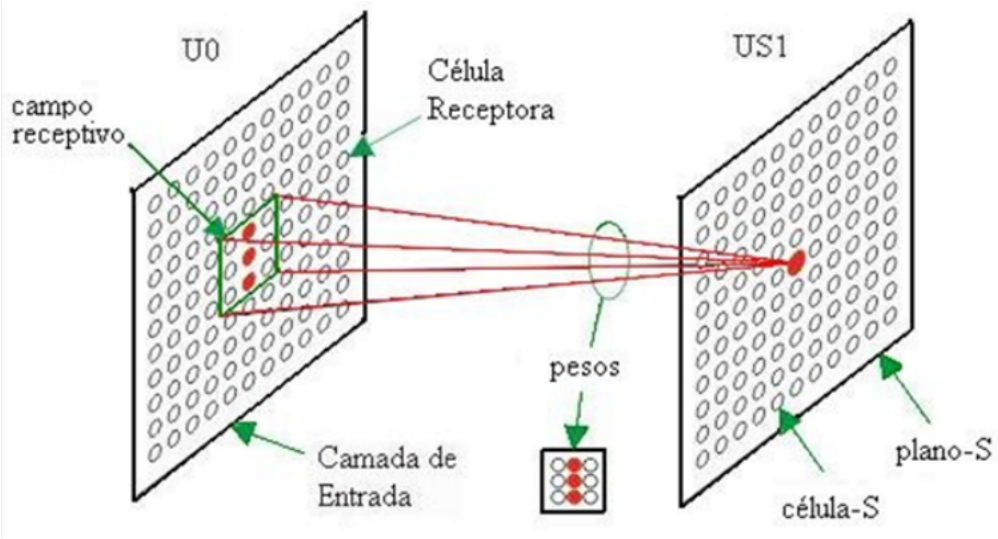

Figura 3.11: Processo de reconhecimento de padrão da célula-S

O valor de saída da Célula-S é determinado pela equação 3.2:

$$u_{S_l}(n,k_l) = \frac{\theta}{1-\theta}.\varphi\left[\frac{1+\sum_{k_{l-1}=1}^{K_{l-1}}\sum_{i\in S_l} a_l(k_{l-1},i,k_l).u_{c_{l-1}}(n+1,k_{l-1})}{1+\theta.b_l(k_l).u_{c_l}(n)} - 1\right] \tag{3.2}$$

O elemento θ é o parâmetro com o qual é possível modificar a habilidade para a célula-S extrair uma determinada característica. O peso-a deve ser maior ou igual a zero, assim como o peso-b e a função de ativação $\varphi[x] = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$.

As Células-S possuem a habilidade de extrair características não apenas treinadas, mas também suas representações deformadas, ou generalizadas. Esta capacidade é influenciada pela escolha do parâmetro θ, chamado limiar (*threshold*). Isso é possível verificar, pois o limiar θ multiplica o valor ponderado provindo da célula-V, no denominador do argumento. Assim, quanto menor o valor de θ maior a capacidade de generalização das características treinadas.

Célula-C

A Célula-C avalia os valores de saída de um Plano-S da Camada-S anterior (Figura 3.12). O valor da Célula-C depende do grau de atividade das Células-S da sua área de conexão. Quanto maior o número de Células-S ativas maior será a atividade da Célula-C. A equação que descreve as Células-C é descrita pela Equação 3.3.

$$u_{C_l}(n,k_l) = \psi\left[\sum_{i\in S_l} d_l(i).u_{S_l}(n+i,k_l)\right] \tag{3.3}$$

Sendo o peso-d ≥ 0 e $\psi[x] = \frac{x}{1+x}$ é $\psi[x] = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$.

Para uma Célula-C ser ativa é suficiente que uma única Célula-S esteja ativa dentro da área de conexão. O Plano-C contem uma representação borrada do Plano-S. Além disso, como o borramento resulta em células-C adjacentes com valores muito próximos, um número reduzido de células-C é necessário para o estágio seguinte. Isso resulta na redução do tamanho do plano-C, em relação ao plano-S, Figura 3.12.

Peso-a

O peso-a é modificado pelo processo de treinamento. Estes pesos são usados nas conexões das Células-C para as Células-S. As características a serem extraídas pelas Células-S estão codificadas nos pesos-a. Estes pesos são ajustados durante o processo de treinamento segundo padrões a serem

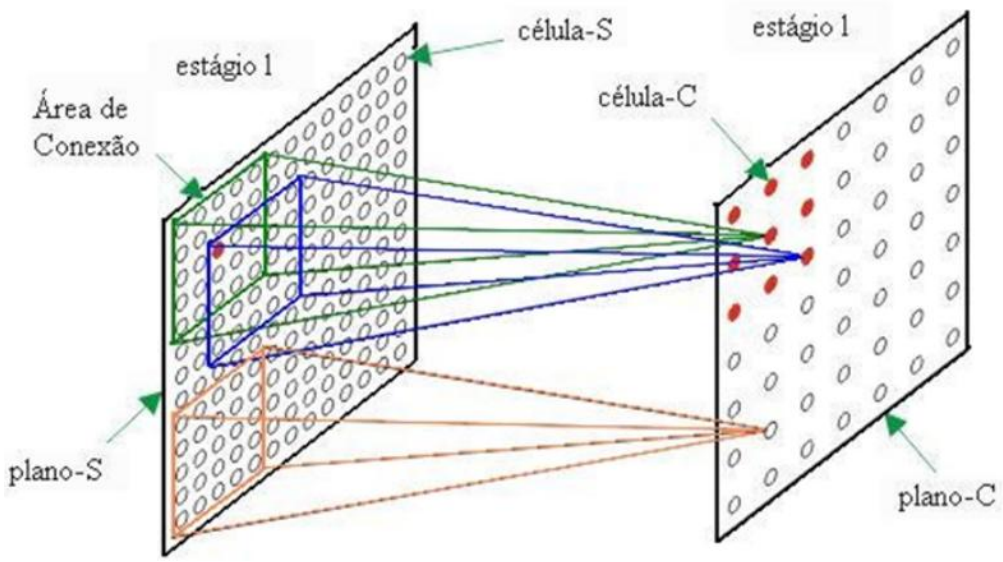

Figura 3.12: Processo de ativação da célula-C

reconhecidos e sua utilização pode ser observada na Figura 3.13.

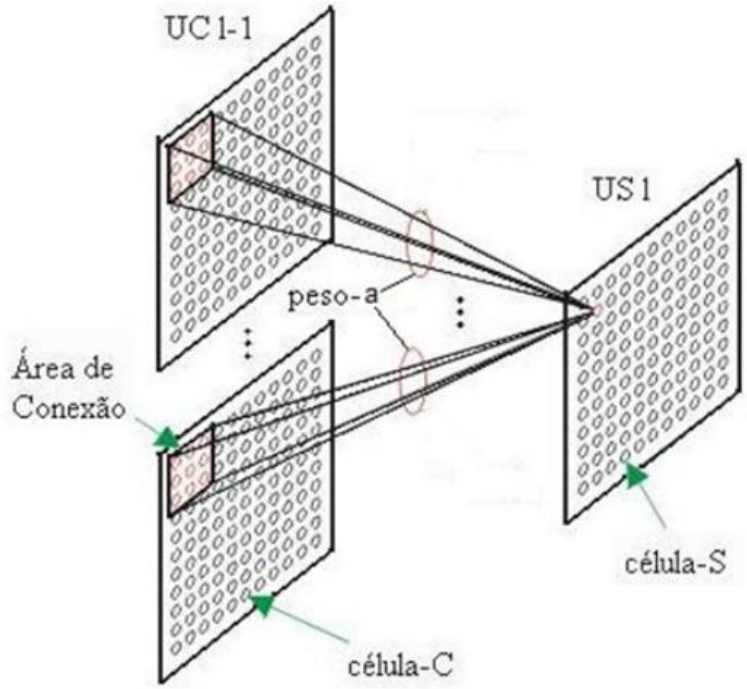

Figura 3.13: Utilização do peso-a

Peso-b

Os pesos-b são também modificados durante o treinamento. Estes pesos são usados nas conexões das Células-V e suas correspondentes Células-S, suas utilizações podem ser observadas na Figura 3.14.

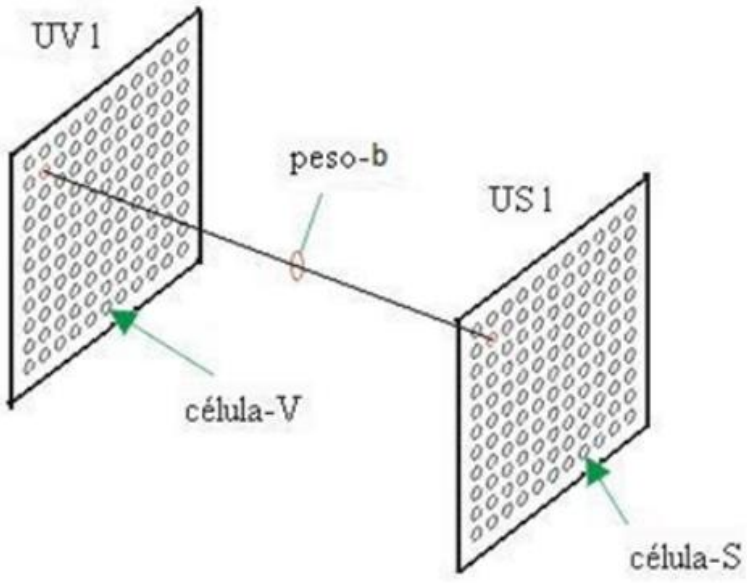

Figura 3.14: Utilização do peso-b

Peso-c

Pesos-c são fixos e são usados nas conexões das Células-C para as Células-V, seus valores são determinados no momento da construção da rede. Estes pesos são configurados de forma a reduzir a transferência de informação das conexões periféricas, ou seja, os pesos são reduzidos monotonicamente a partir do centro da área de conexão. Além disso, a soma de todos os pesos-c das conexões de entrada de uma célula-V deve ser igual a 1; sua utilização pode ser observada na Figura 3.15.

Peso-d

Pesos-d são fixos e são utilizados nas conexões das Células-S para Células-C. Assim como os pesos-c, os pesos-d são determinados no momento da construção da rede e também têm os valores reduzidos monoticamente a partir do centro da área de conexão, como pode ser observado na Figura 3.16.

3.2.3 Treinamento da Rede

Apesar de existirem diversos modos de treinamento para a rede Neocognitron, descreve-se aqui o método projetado originalmente, que é o aprendizado sem supervisão.

A princípio, o treinamento segue conforme a maioria das redes neurais, ou seja, apresenta-se uma amostra à rede, e os dados são propagados pela rede, permitindo que os pesos das conexões

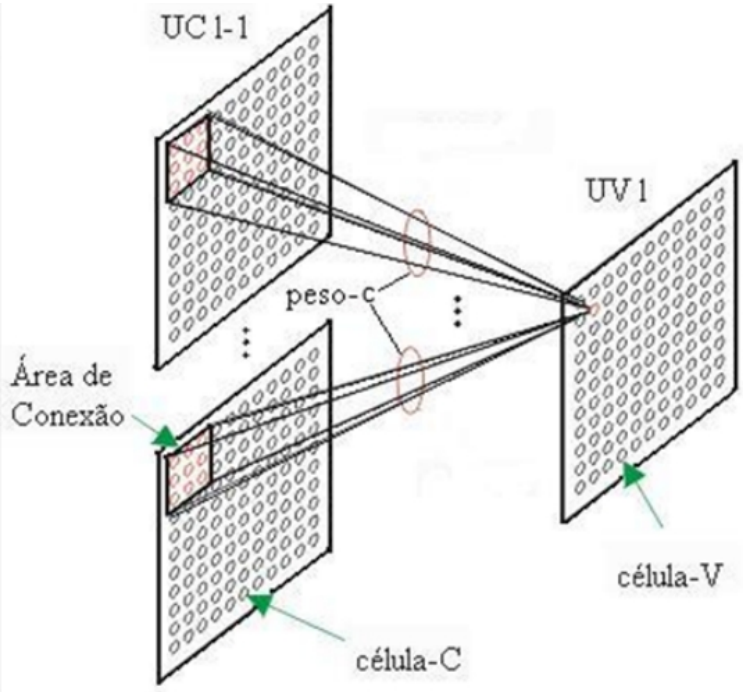

Figura 3.15: Utilização do peso-c

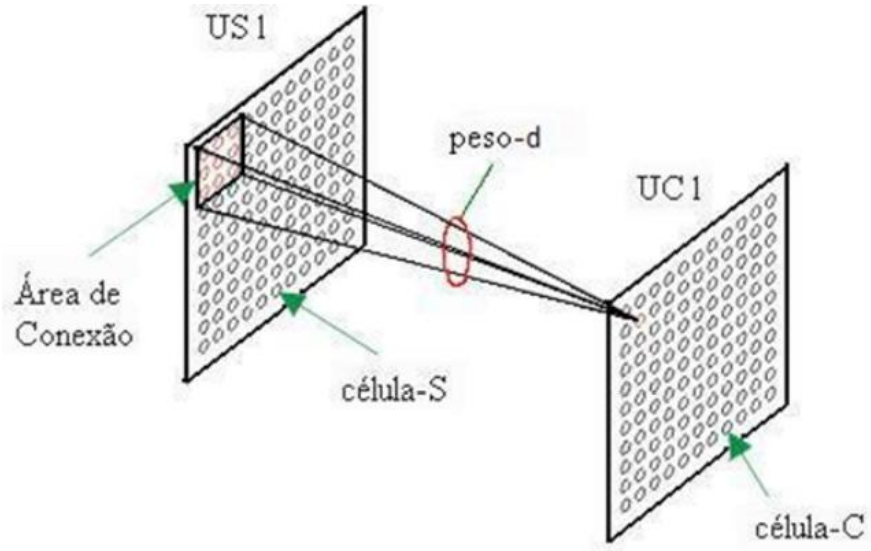

Figura 3.16: Utilização do peso-d

se ajustem progressivamente de acordo com um algoritmo determinado. Depois dos pesos serem atualizados, é apresentado um segundo padrão na camada de entrada, e o processo se repete com todas as amostras de treinamento, até que a rede classifique corretamente os padrões.

A rede Neocognitron possui a característica de que todas as células em um mesmo plano compartilham o mesmo conjunto de pesos. Portanto, apenas uma única célula de cada plano precisa participar do treinamento, e, após isso, distribuir o seu conjunto de pesos para as demais células.

Para entender melhor o funcionamento, pode-se imaginar todos os Planos-S de uma camada

empilhados uns sobre os outros, alinhados de tal modo que as células correspondentes a uma determinada localidade estejam diretamente umas acima das outras. Com isso, consegue-se imaginar diversas colunas, cortando perpendicularmente os planos. Essas colunas criam grupos de Células-S, onde todos os membros do grupo têm campos receptivos na mesma localidade na camada de entrada.

Com esse modelo em mente, pode-se agora aplicar um padrão de entrada e examinar a resposta das Células-S em cada coluna. Para garantir que cada Célula-S forneça uma resposta distinta, pode-se iniciar os pesos a_l com valores aleatórios pequenos e positivos e os pesos inibidores b_l com zero. Primeiro, anota-se o plano e a posição da Célula-S cuja resposta é a mais forte em cada coluna. Então examina-se os planos individualmente de modo que, caso um plano possua duas ou mais dessas Células-S, escolhe-se somente a Célula-S com a resposta mais forte, sujeita à condição de que cada uma das células esteja em uma Coluna-S diferente.

Essas Células-S se tornam os protótipos, ou representantes, de todas as células em seus respectivos planos. Uma vez escolhidos os representantes, as atualizações dos pesos são feitas de acordo com as Equação 3.4 e Equação 3.5:

$$\Delta a_l(k_{l-1}, v, k_l) = q_l c_{l-1}(k_(l-1), n+v) \tag{3.4}$$

$$\Delta b_l(k_l) = q_l v_{c_{l-1}}(n) \tag{3.5}$$

Com esse algoritmo, uma vez que as células de um plano passem a responder a uma determinada característica, elas passam a emitir respostas menores em relação a outras características.

Algoritmos

A seguir são descritos os algoritmos de simulação da rede neocognitron em pseudo-linguagem computacional. O Algoritmo 3.1 é usado para a computação das células-S e células-C de planos celulares k_l do estágio.

Algoritmo 3.1: Algoritmo para computar um plano celular

```
Procedure Computar_Plano S e C
        Begin
        ;Computa Plano-S
        For n = 1 to N do Begin
5               For k_{l-1} = 1 to K_{l-1} do begin
                        For all v $\in$ Sv do begin
```

```
                                    e(n,k_l) := e(n,k_l) + a(k_{l-1},v,k_l).u_{c_{
                                        l-1}}(n+v,k_{l-1});
                                    h(n,k_l) := h(n,k_l) + c(v).{u_{c_{l-1}}(k_{l
                                        -1},n+v)}^2;
                            End For
                            U_{v_l}(n,k_l) := $\sqrt{h(n,k_l)}$;
                            U_{S_l}(n,k_l) := ($\theta$/(1-$\theta$)).$\varphi$
                                ((1+e(n,k_l))/(1+$\theta$.b(k_l).u_{v_l}(n,k_l)-1)
                                ;
                    End For
            End For
            For n = 1 to N do begin
                    For all v $\in$ S_v do
                                    U_{c_l}(n,k_l) := u_{c_l}(n,k_l)+d(v).u_{s_l}(
                                        n+v,k_l);
                            U_{c_l}(n,k_l) := $\Psi$(u_{c_l}(n,k_l));
                    End for
            End For
End
```

O Algoritmo 3.1 computa valores $u_{S_l}(n,k_l)$ de células-S e os valores u_{C_l} de células-C, dentro de um estágio l. São computados esses valores para todas as N posições dentro de um plano celular. Para cálculos do valor $u_{S_l}(n,k_l$ são computados os somatórios $e(n,k_l)$ e $h(n,k_l)$ de todas as entradas conectadas aos K_{l-1} planos celulares da camada precedente, numa dada área de conexão S , que circunda a posição da célula n da camada de célula-C, do estágio anterior ou da camada de entrada, pela interação sobre os comandos $c1$ e $c2$.

$$e(n,k_l) = e(n,k_l) + a(v,k_{l-1},k_l).u_{C_{l-1}}(n+v,k_{l-1}) \quad \text{(c1)}$$
$$h(n,k_l) = h(n,k_l) + c(v).u_{C_{l-1}}(k_{l-1},n+v)^2 \quad \text{(c2)}$$

Chamando de $u_{Cl}(m,k_l)$ a raiz quadrada de $h(n,k_l)$, dada por $\sqrt{h(n,k_l}$, tem-se o valor $u_{S_l}(n,k_l)$, obtido pelo comando $c3$.

$$u_{S_l}(n,k_l) = \left(\frac{\theta}{(1-\theta)}\right).\psi\left(\frac{1+e(n,k_l)}{1+\theta b(k_l).u_{V_l}(n,k_l)} - 1\right) \quad \text{(c3)}$$

Onde $\psi(x) = x$, quando $x > 0$, e $\psi(x) = 0$, caso contrário. A variável θ representa o limiar da função, cujos valores ficam entre 0 e 1, $b(k_l)$ representa o coeficiente de inibição.

Para se obter o valor $u_{C_l}(n,k_l)$, é computado primeiramente o somatório de todas as entradas correspondentes aos valores $u_{S_l}(n,k_l)$, previamente obtidos em uma área de conexão , que circunda a posição da camada de célula-S precedente, pela interação sobre o seguinte comando $c4$.

$$u_{C_l} = u_{C_l}(n,k_l) + d(v).u_{S_l}(n+v,k_l) \quad \text{(c4)}$$

Seguida do cálculo da função de transferência $\psi(x) = 1/(1+x)$, que limita a saída das células-C no intervalo $[0,1]$, ou seja, $u_{C_l} = \psi(u_{C_l}(n,k_l))$.

A fase de treinamento da rede utiliza o Algoritmo 3.2.

Algoritmo 3.2: Treinamento de um estágio da rede Neocognitron

```
Procedure Treinar_Estagio(l)
Begin
        ; fica repetindo ate que todos os fatores do
        ; do padrao de entrada tenham sido encontrados
        Repeat
                For k_l = 1 to K_{l+1} do computar_plano(l);
                        selecionado := false;
                Repeat
                        If proximo_vencedor > 0 then
                        Begin
                                vencedor        := proximo_vencedor;
                                selecionado     := true;
                                For k_l = 1 to K_l do
                                        If us(vencedor,k_l) > 0 then
                                                selecionado := false;
                        End;
                Until (selecionado or proximo_vencedor = 0);
                If selecionado then
                Begin
                        For k = 1 to K_{l-1} do
                                For all v \in S do
                                        ; reforcar os pesos na area de conexao
                                        a(k_{l-1},v,K_l) := a(k_{l-1},v,k_l) +
                                            q.c(v).u_{cl-1}(vencedor+v,k_l);
                        b(k_l) := b(k_l) + q . sqrt(h(vencedor,k_l));
                        K_l := K_l + 1;
                End;
        Until not (selecionado);
End;
```

O Algoritmo 3.2 mostra o treinamento dentro de um estágio que consiste em acrescentar novos planos celulares a cada novo fator ou característica extraída, incrementando assim o número K_l de planos celulares. O treinamento da rede Neocognitron procede computando os valores de células-S para os planos celulares $k_l = 1$ a $k_l = K_l$, relacionados aos fatores já existentes, e $k_l = K_l + 1$, que corresponde ao plano de seleção de semente (SSP - *Seed Selection Plane*). O plano de seleção de semente é um plano auxiliar para a obtenção de um novo plano celular, correspondente a uma nova

característica, cujos pesos iniciais das conexões de entrada são valores pequenos, porém diferentes de zero, para que seus neurônios possam ser excitados para quaisquer fatores apresentados nos padrões de entrada.

Então, é verificado nesse último plano, plano de seleção de semente, a célula vencedora (*winner*), ou seja, aquele em que o valor de resposta é máximo. Se na posição da célula vencedora existe alguma resposta maior que zero nos K_l planos celulares anteriores já treinados (fatores já existentes), o algoritmo segue na procura de um novo vencedor; caso contrário, cada conexão de entrada da célula vencedora é reforçada proporcionando à intensidade da conexão de entrada por meio dos comandos $c5$ e $c6$:

$$a(k_{l-1}, v, k_l) = a(k_{l-1}), v, k_l) + q.c(v).u_{C_l}(vencedor + v, k) \quad \text{(c5)}$$

$$b(k_l) = b(k_l) + q.\sqrt{h(vencedor), k_l} \quad \text{(c6)}$$

Desta forma, um novo fator é obtido e um novo plano celular é adicionado à camada, incrementando K_l de um. Uma vez feito o reforço das conexões de entrada na célula vencedora, essa célula passa a ser a célula semente, pois todas as demais células do mesmo plano celular terão os mesmos pesos nas conexões de entrada. Daí surge o nome *seed-selection-plane* ao plano usado para obter a célula vencedora.

O procedimento de treinamento da camada-S, descrito acima, é repetido até que todos os novos fatores sejam detectados, com a apresentação dos padrões de treinamento na camada de entrada.

3.2.4 Conclusão

A rede Neocognitron é uma rede neural que simula o sistema visual dos mamíferos. Por este fato possui uma grande capacidade de reconhecimento de padrões e de aprendizado de formas complexas. Esta possui ainda um limiar que regula sua capacidade de generalização do reconhecimento do padrão de entrada.

No próximo capítulo apresenta-se a base teórica da arquitetura de computação de alto desempenho, para a implementação paralela da rede Neocognitron.

4 Computação de Alto Desempenho

Visando atender a demanda crescente de processamento de alto desempenho, têm sido desenvolvidos vários modelos de arquiteturas paralelas. A maioria dessas arquiteturas é caracterizada pela presença de múltiplos processadores que cooperam entre si. No entanto, existem outros modelos de arquitetura paralelas, motivo pelo qual é feita a descrição seguinte geral desses modelos.

4.1 Arquiteturas Paralelas

Michael Flynn (FLYMN, 1972)classificou as arquiteturas paralelas de acordo com o fluxo de dados e instruções. Dependendo se estes fluxos são múltiplos ou não e por meio da combinação das possibilidades, Flimn propôs 4 categorias: SISD, SIMD, MISD e MIMD.

4.1.1 SISD

Essa categoria consiste na existência de um único fluxo de instruções e um único fluxo de dados, caracterizando uma arquitetura com um único processador, como mostra a Figura 4.1, onde se vê o fluxo de instruções partindo da memória M passando por uma unidade de controle (UC), para ser então processado (P) e tendo os dados sendo lidos ou gravados em uma posição de memória (M).

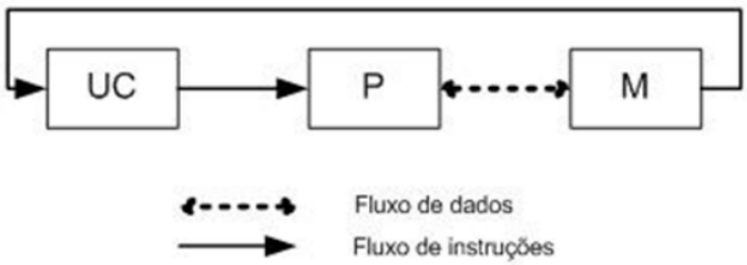

Figura 4.1: Diagrama de blocos de uma arquitetura SISD

Um exemplo de fluxo de instruções a serem executadas nessa arquitetura pode ser observada

na Figura 4.2, onde observa-se que a cada momento (tempo) existe apenas uma instrução sendo executada.

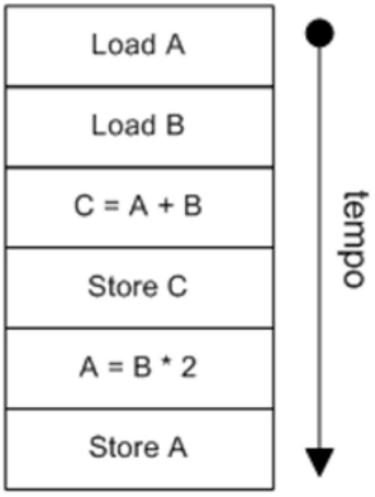

Figura 4.2: Fluxo de instruções sendo executadas na arquitetura SISD

4.1.2 SIMD

Essa categoria consiste na existência de um fluxo único de instruções, porém que corresponde à manipulação de um fluxo múltiplo de dados sobre os quais se realizam operações idênticas em elementos de processamento distintos, que operam sincronizadamente entre si, como pode-se ver na Figura 4.3.

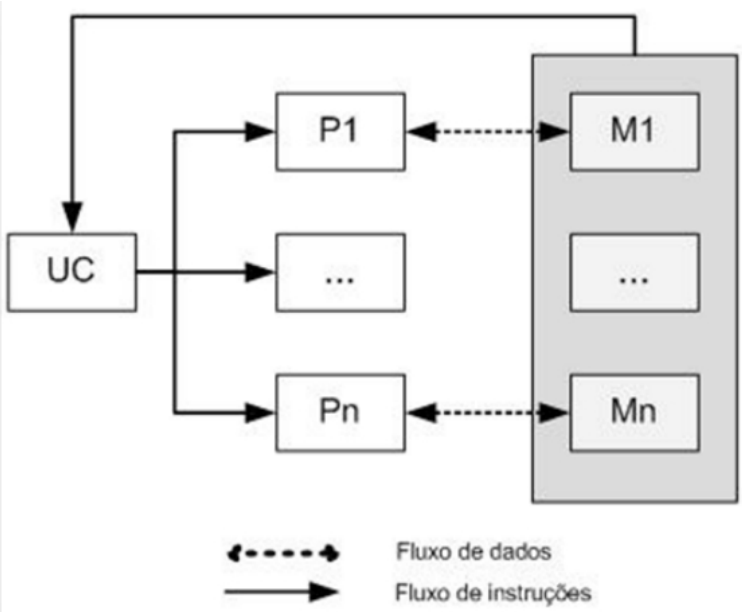

Figura 4.3: Diagrama de blocos de uma arquitetura SIMD

Na Figura 4.4 pode-se ver um exemplo de um fluxo de instruções sendo executadas em diferentes processadores ao mesmo tempo.

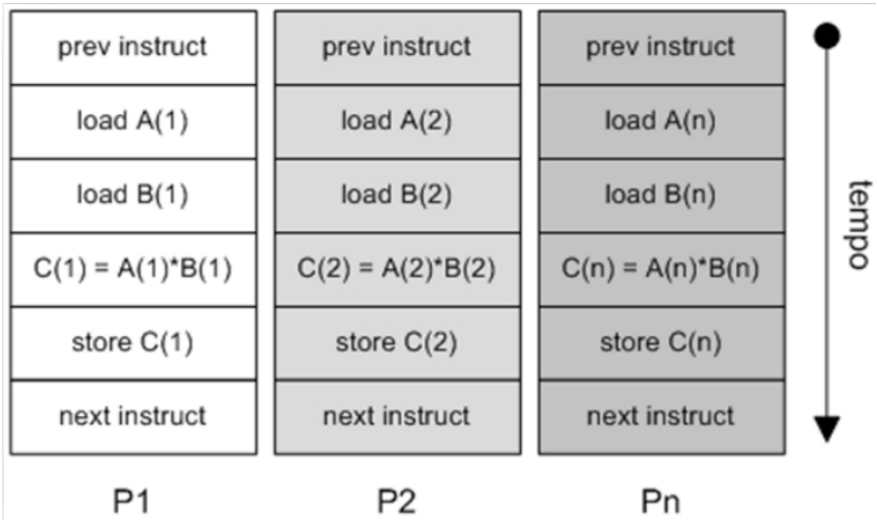

Figura 4.4: Exemplo de fluxo de instruções na arquitetura SIMD

4.1.3 MISD

Essa categoria consiste na existência de fluxo múltiplo de instruções sobre fluxo único de dados, como pode ser observado na Figura 4.5.

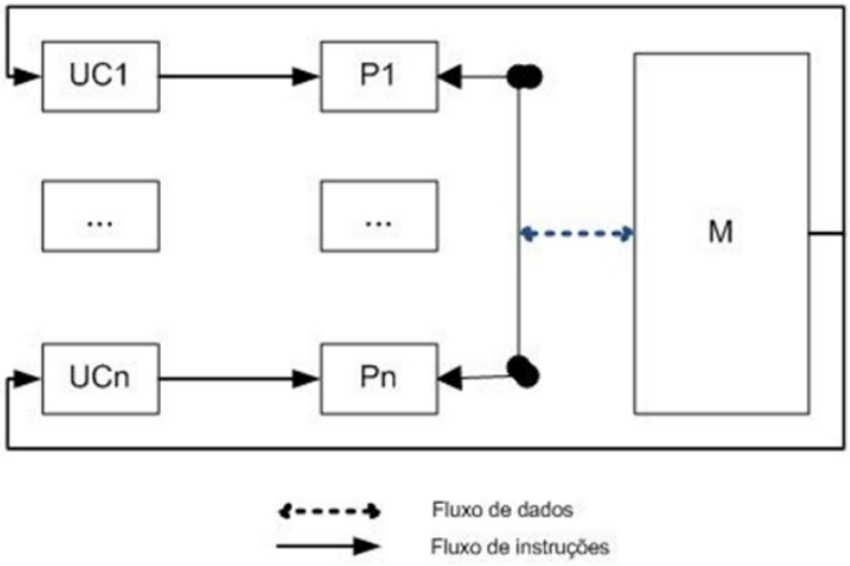

Figura 4.5: Diagrama de blocos de uma arquitetura MISD

As máquinas com arquitetura MISD são classificadas diferentemente segundo vários autores. Para alguns, correspondem a processadores vetoriais; outros descrevem-nas como variantes de máquinas SIMD e para a maioria, porém, tais máquinas não têm existência prática.

4.1.4 MIMD

Essa categoria consiste na existência de fluxo múltiplo de instruções sobre fluxo múltiplo de dados como pode ser observado na Figura 4.6.

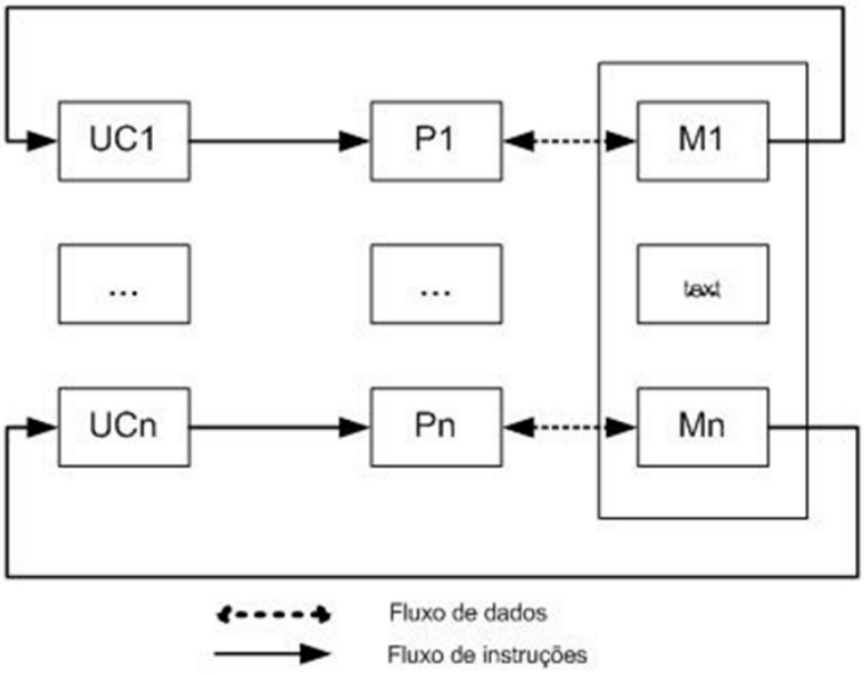

Figura 4.6: Diagrama de blocos de uma arquitetura MIMD

Na Figura 4.7 pode ser obsevado um exemplo de um conjunto de fluxo de instruções sendo executadas em uma arquitetura MIMD, para cada processador (Px) existe um conjunto distinto de fluxo de instruções a serem executadas.

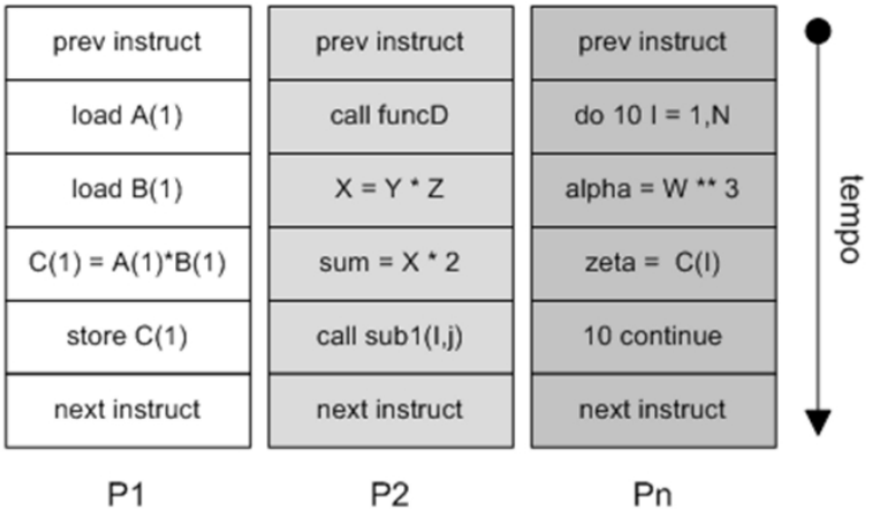

Figura 4.7: Fluxo de instruções sendo executadas na arquitetura MIMD

Este tipo de arquitetura possui a vantagem dos processadores poderem executar várias instru-

ções simultaneamente. Já, em contrapartida, possui a desvantagem do custo de balanceamento de carga.

4.2 Programação Paralela

As seções anteriores apresentaram as principais classes de arquiteturas paralelas. Para que este ambiente funcione de forma adequada, dentro de um padrão de qualidade e necessidades do projeto a ser desenvolvido é necessário que este seja implemento dentro de uma filosofia de programação paralela.

Nesta sessão apresenta-se os conceitos que envolvem o desenvolvimento de software para serem executados dentro de ambientes paralelos.

4.2.1 Conceitos Básicos Sobre Programação Paralela

Paralelismo é uma técnica usada em tarefas grandes e complexas para obter resultados em tempo desejável, dividindo-as em tarefas menores, executadas em vários processadores para serem executadas simultaneamente, como é apresentado na Figura 4.8.

O termo paralelismo em computação é normalmente utilizado para designar paralelismo físico, ou seja, o fato de se ter múltiplos elementos de processamento para executar os processos, sendo que todos eles podem estar ativos simultaneamente (HWANG; XU, 1998).

A concorrência implica em mais de um processo iniciado e não terminado. Não existe uma relação com o número de elementos de processamento utilizados. Obter paralelismo na execução desses processos só é possível quando existe mais de um elemento de processamento, de modo que diversos processos possam estar em execução em um determinado instante.

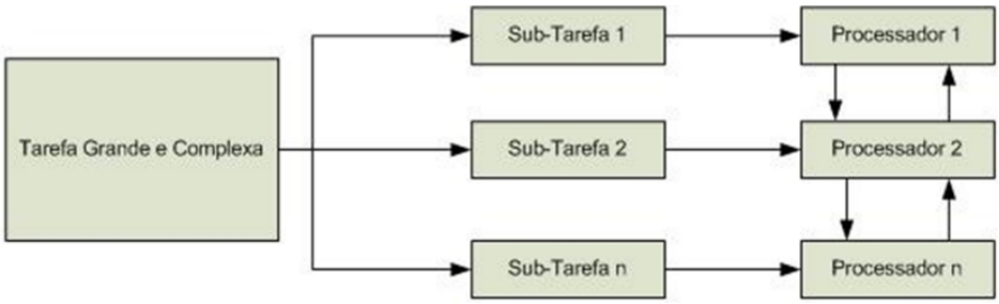

Figura 4.8: Diagrama de Paralelização

Quando existe apenas um elemento de processamento e vários processos estão sendo executados de maneira concorrente existe um pseudo-paralelismo. O usuário tem a impressão de que os

processos estão sendo executados ao mesmo tempo, mas o que realmente acontece é o compartilhamento do elemento de processamento entre os processos em execução. Isto é, em um determinado instante de tempo, apenas um processo está em execução, enquanto os demais estão aguardando a liberação do processador ou de outro recurso.

Os principais objetivos do paralelismo são:

- Aumentar o desempenho (reduzindo o tempo) no processamento;
- Fazer uso de um sistema distribuído para a resolução de tarefas e
- Obter ganhos de desempenho.

4.2.2 Tipos de Paralelismo Quanto ao Conteúdo

Quanto ao conteúdo a ser paralelizado, pode-se fazer a seguinte classificação:

Paralelismo de Dados

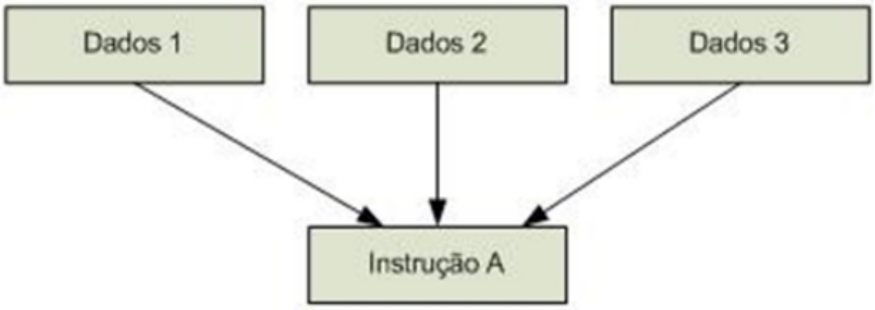

Figura 4.9: Paralelismo de dados

Neste tipo de paralelismo, o processador executa as mesmas instruções sobre dados diferentes, como mostra a Figura 4.9. Aplicado, por exemplo, em programas que utilizam matrizes imensas e para cálculos de elementos finitos. Exemplos: resolução de sistemas de equações e multiplicação de matrizes.

Paralelismo Funcional

Neste caso, o processador executa instruções diferentes que podem ou não operar sobre o mesmo conjunto de dados, como mostra a Figura 4.10. Aplicado em programas dinâmicos e modulares onde cada tarefa é um programa diferente. Exemplos: Paradigma Produto-Consumidor, simulação e rotinas específicas para tratamento de dados (imagens).

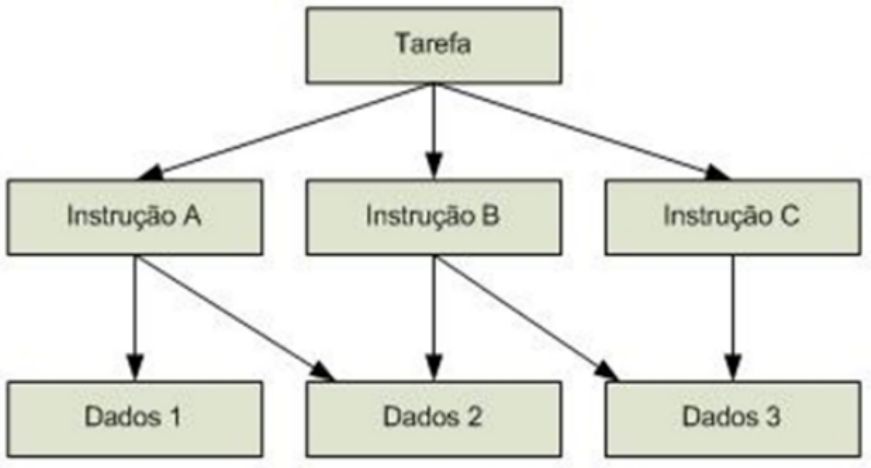

Figura 4.10: Paralelismo functional

Paralelismo de Objetos

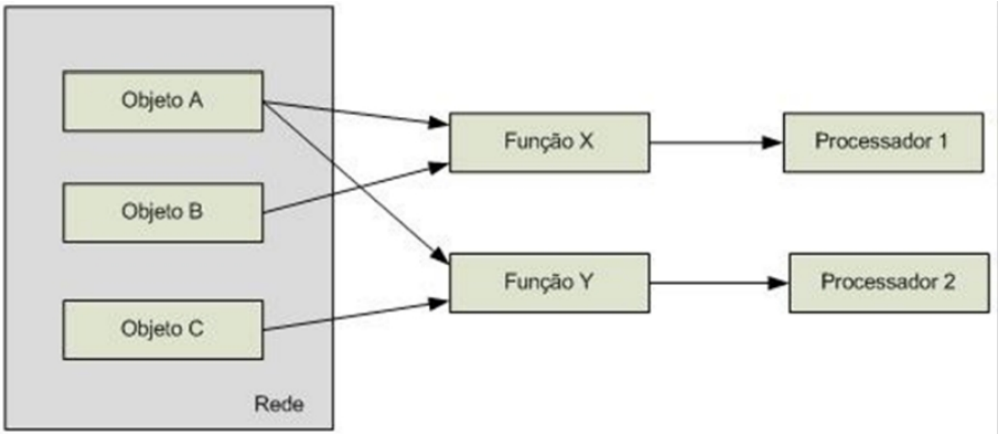

Figura 4.11: Paralelismo de Objetos

Este é o modelo mais recente, que utiliza o conceito de objetos distribuídos por uma rede (como a Internet), capazes de serem acessados por métodos (funções) em diferentes processadores para uma determinada finalidade, como mostra a Figura 4.11. Exemplo: MPP.

4.2.3 Tipos de Paralelismo Quanto a Forma

A escolha de um paradigma é fundamental na elaboração de um algoritmo paralelo, para a obtenção do desempenho desejado. Existem 5 tipos de paradigmas (HWANG; XU, 1998):

- Fases Paralelas (Phase Parallel)
- Divisão e Conquista (*divide and conquer*)

- Pipeline
- Mestre/Escravo (*Processor Farm*) e
- *Pool* de trabalho (*work pool*)

Esses paradigmas são descritos a seguir.

Fases Paralelas

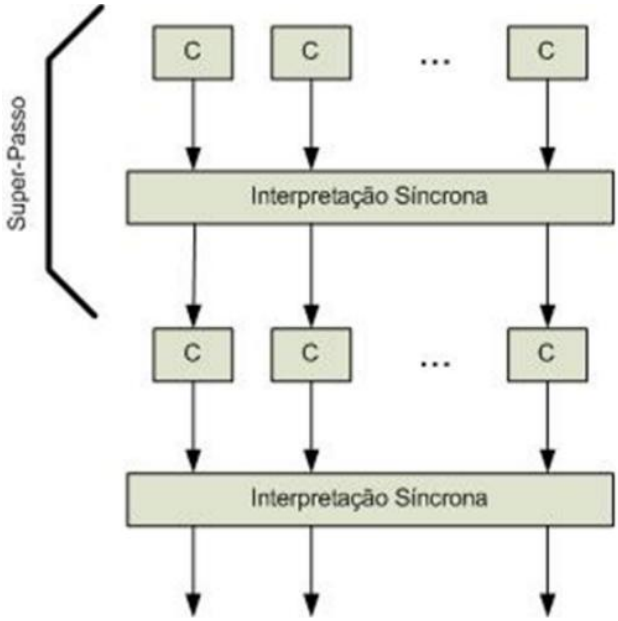

Figura 4.12: Fases Paralelas

Nesse paradigma, cujo diagrama é apresentado na Figura 4.12, o programa paralelo consiste de um número de super passos e cada super-passo tem duas fases: a fase computacional (C) e a fase de interação, que consiste da sincronização e comunicação. Na fase computacional, múltiplos processos desempenham individualmente uma computação independente C. Na fase de interação subseqüente, os processos desempenham uma ou mais operações de interações síncronas, tais como um barrier ou uma comunicação de bloco. Então, o próximo super-passo é executado.

Esse tipo de paradigma facilita a detecção de erros e a análise de desempenho, mas tem duas principais falhas: congestionamento nas comunicações e a dificuldade em manter o balanceamento da carga de trabalho entre os processos.

Divisão e Conquista

O paradigma paralelo *divide and conquer*, cujo diagrama é mostrado na Figura 4.13, é muito similar a um sistema seqüencial. Um processo pai divide as cargas de trabalho em várias partes

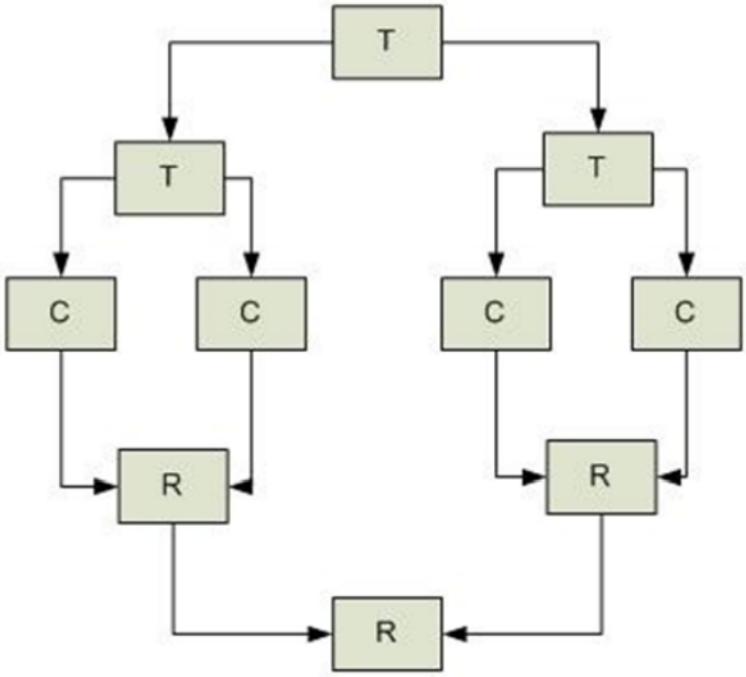

Figura 4.13: Divisão e Conquista

pequenas e as atribui para um número de processos filhos.

Os processos filhos, então, computam suas cargas de trabalho em paralelo e os resultados são agrupados pelos pais. A divisão e os procedimentos de agrupamento são feitos recursivamente. Este paradigma é muito natural para computação, mas sua desvantagen é a dificuldade em realizar o balanceamento de carga.

Pipeline

Neste tipo de paralelismo, a aplicação é constituída de uma seqüência de dados que recebem o processamento em forma de tarefas específicas que devem ser executadas seqüencialmente por processadores distintos, como mostra o diagrama da Figura 4.14.

Pode-se fazer uma analogia com a construção de uma casa, onde são necessários vários especialistas, como eletricista, engenheiro, encanador, arquiteto, entre outros. Cada um destes tem uma determinada tarefa e alguns têm que esperar o término de outra tarefa para começar a sua, ou seja, cada um é responsável pela sua tarefa e todos trabalham em paralelo, desde que hajam várias casas para serem construídas.

Este paradigma é o mais natural para se desenvolver em paralelo tendo como base um programa seqüencial, sendo até possível a sua detecção por compiladores e analisadores de código fonte. Dentre as desvantagens pode-se citar:

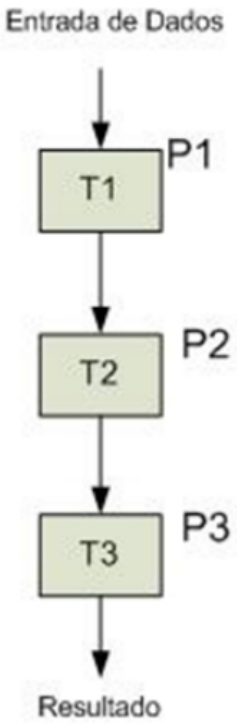

Figura 4.14: *Pipeline*

- Pouca flexibilidade, visto que modificações no algoritmo podem requerer mudanças drásticas na rede de processadores
- O balanceamento de carga deve ser feito de maneira cuidadosa a fim de que as tarefas específicas lentas não tornem a execução total lenta e
- A relação entre comunicação e execução deve ser baixa a fim de que a sobrecarga de comunicação não torne o processamento ineficiente.

Mestre/Escravo

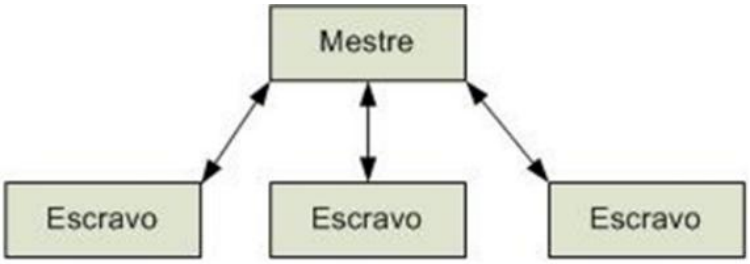

Figura 4.15: Mestre/Escravo

Neste paradigma, um processador é denominado "mestre"e os demais processadores são denominados "escravos", como mostra na 4.15. O processamento consiste de um conjunto de tarefas

que são distribuídas pelo processador "mestre"aos processadores "escravos". O processamento "mestre"supervisiona os trabalhos dos "escravos", cada um processando assincronamente as respectivas tarefas.

Este modelo de paralelismo possui várias vantagens como:

- Facilidade de paralelismo do sistema, o que pode ser conseguido por meio do aumento do número de processadores;
- Facilidade de programação e
- Balanceamento de carga mais natural, visto que as tarefas vão sendo submetidas aos processadores de acordo com a disponibilidade.

Como desvantagens, pode-se citar a sobrecarga de comunicação e a possibilidade de gargalo no processador "mestre".

Pool de Trabalho

Este paradigma é análogo a uma comunidade disposta a executar um trabalho constituído de uma série de tarefas relacionadas numa lista. O paradigma representado pelo diagrama da Figura 4.16 é freqüentemente usado no modelo de variável compartilhada. Um conjunto de trabalhos é realizado uma estrutura de dados global.

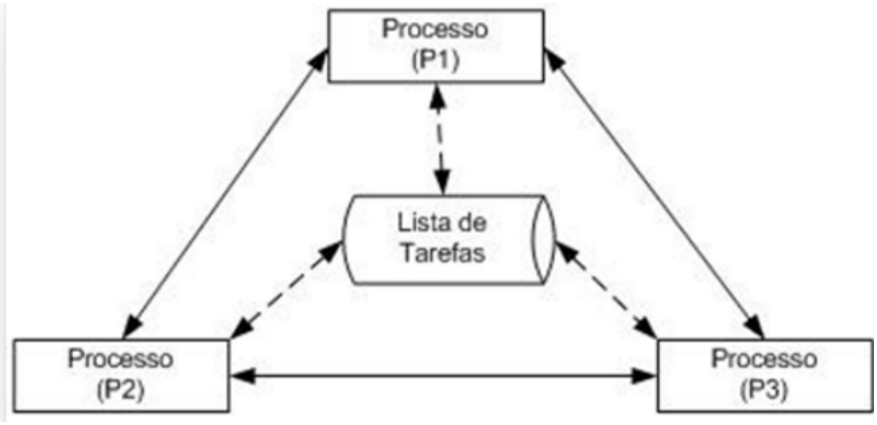

Figura 4.16: *Pool* de trabalho

Inicialmente, pode-se ter somente uma tarefa na lista. Algum processador livre busca essa tarefa e a executa, produzindo zero, uma ou mais novas tarefas que são colocadas na lista. O programa paralelo finaliza quando a lista torna-se vazia.

Este paradigma facilita no balanceamento de carga, pois a carga de trabalho é dinamicamente alocada para processadores; entretanto, a implementação do modelo de forma a conseguir eficiência, não é fácil, especialmente usando passagem de mensagem.

4.3 Exploração do Paralelismo

Existem alguns problemas a serem analisados para a exploração do paralelismo, pois a eficiência na execução paralela depende do particionamento em módulos dos programas e no escalonamento desses módulos entre os processadores.

Com relação ao problema do particionamento de um programa em tarefas paralelas, procura-se identificar as partições para a obtenção do menor tempo possível na execução, e conseqüentemente, o tamanho adequado para cada uma destas tarefas. Denomina-se o problema do particionamento como a análise da granularidade de tarefas para processamento paralelo.

Pode-se citar duas abordagens para o problema de particionamento de um programa em tarefas a serem executadas paralelamente (MCCREARY; GILL, 1989):

- Paralelismo Explícito: é a detecção do paralelismo pelo programador. Nesta abordagem o programador utiliza linguagens especiais ou bibliotecas para troca de informações entre os processadores. O paralelismo explícito contribui para a diminuição da complexidade dos compiladores para sistemas paralelos pois elimina a necessidade da detecção automática do paralelismo em tempo de compilação.
- Paralelismo Implícito: é a detecção automática do paralelismo. Nesta abordagem, os compiladores devem detectar o paralelismo inerente ao programa desenvolvido nas linguagens seqüenciais convencionais.

4.4 Ambiente Paralelo

O ambiente paralelo pode ser definido como o conjunto de hardware e software para possibilitar o desenvolvimento do processamento paralelo. Um exemplo de ambiente paralelo é a união de vários processadores interligados em rede, como uma plataforma para manipulação de processos paralelos, com um sistema operacional, e uma linguagem de programação, usando um dos modelos de programação paralela.

4.4.1 Granularidade

Granularidade, ou nível de paralelismo, indica a relação entre o tamanho de cada tarefa e o tamanho total do programa, ou seja, é a razão entre computação e comunicação. A granularidade pode ser dividida em três níveis (HWANG; XU, 1998) (ALMASI; GOTTLIEB, 1994) (KIERNER, 1991) (NAVAUX, 1989) (HWANG; BRIGGS, 1984):

- Fina: relaciona o paralelismo no nível de instruções, ou operações, e implica grande número de processadores pequenos e simples.
- Média: situa-se em patamar entre as duas anteriores, implicando procedimentos executados em paralelo.
- Grossa: relacionada com o paralelismo no nível dos processos e geralmente se aplica a plataformas com poucos processadores grandes e complexos.

4.4.2 Construção de Algoritmos Paralelos

Para se construir um algoritmo paralelo, há vários itens que devem ser analisados (CENTURION, 1998), tais como:

- O modelo de paralelismo a ser adotado em função das características da aplicação;
- A forma de distribuição das tarefas nos processadores, para obter um melhor desempenho;
- A ferramenta usada no desenvolvimento do algoritmo paralelo;
- A linguagem a ser usada na implementação do algoritmo e
- Arquitetura na qual se executará o algoritmo, visto que a sua eficiência pode variar de maneira drástica de acordo com o tipo de arquitetura.

Ou seja, para a construção de um algoritmo paralelo é necessário escolher qual a melhor abordagem dentre: começar de um algoritmo seqüencial disponível, já implementado ou a partir das especificações do problema a ser resolvido com a computação paralela.

Ao definir o algoritmo paralelo, deve-se identificar os processos que serão executados em paralelo e verificar se a divisão é conveniente, considerando as comunicações entre processos e tamanho dos processos necessários. Após ter definido os processos, deve-se organizar estes em forma de programas para isto, deve-se conhecer a arquitetura, o algoritmo proposto e o modelo de paralelismo.

4.5 Implementação de Programas Paralelos

Para se expressar o paralelismo durante a implementação do programa deve-se utilizar as formas para ativação de processos paralelos, como a sincronização entre os processos, bem como a comunicação.

4.5.1 Mapeamento

Nessa etapa deve-se fazer a alocação dos processos nos processadores e definir como será a comunicação por meio de:

- Escalonamento: forma como os processos são colocados e retirados dos processadores.
- Balanceamento de Carga: fator considerado para que um processador não seja sobrecarregado enquanto outro está ocioso.
- Migração de Processos: considera a troca de uma tarefa em execução para outro processador em tempo de execução.

4.6 Avaliação do Desempenho

Ao passar por todas as etapas acima, deve-se medir o desempenho do programa. Este desempenho permite analisar o ganho obtido com o aumento do total de processadores utilizados. Algumas medidas são usadas, como a eficiência e o *speed-up* (ganho de desempenho).

Uma característica fundamental da computação paralela é o aumento de velocidade de processamento por meio da utilização do paralelismo. Para se medir o ganho com o aumento de processadores pode-se utilizar algumas medidas como speed-up e eficiência (QUINN, 1987) (SOUZA, 1996).

4.6.1 *Speed-up*

Speed-up é uma medida utilizada para determinar o aumento de velocidade obtido para a execução de um programa utilizando p processadores, em relação Ãă sua execução seqüencial, usando apenas um processador, sendo obtido pela Equação 4.1.

$$S_p = \frac{T_{seq}}{T_{par}} \tag{4.1}$$

O caso ideal é quando o $S_p = p$, isto é, a velocidade de processamento aumenta linear e proporcionalmente à quantidade de processadores utilizados. Mas existem fatores que degradam essa situação ideal:

- A sobrecarga de comunicação
- Algoritmos parcialmente paralelizáveis
- A dificuldade de balanceamento de carga entre os processadores e
- Casos onde a granularidade é inadequada para o tipo de arquitetura utilizada

Existem também casos em que o *speed-up* alcançado é maior que o ideal, isto é, $S_p > p$. Esse fenômeno é chamado de anomalia de *speed-up* ou *speed-up* super linear.

4.6.2 Eficiência

Eficiência é uma medida que mostra o quanto o paralelismo foi explorado no algoritmo e é dado pela Equação 4.2

$$E_p = \frac{S_p}{p} \tag{4.2}$$

A eficiência quantifica a utilização do processador. Variando entre 0 e 1, o caso ideal é verificado quando $E_p = 1$, indicando uma eficiência de 100%.

4.7 Ferramentas para Programação Paralela

As ferramentas para programação paralela devem ser eficientes e de fácil utilização para comunicação e sincronização entre os processos e a ativação de processos paralelos. A escolha do tipo de ferramenta que será utilizada para o desenvolvimento de um programa paralelo é um decisão extremamente importante, devendo levar em consideração vários fatores; entre os quais: a aplicação, o usuário que utilizará a ferramenta e a arquitetura que executará os códigos gerados.

Pode-se citar alguns tipos de ferramentas usadas na construção de programas paralelos: ambientes de paralelização automática, extensões paralelas para linguagens seriais e linguagens concorrentes.

4.7.1 Compiladores Paralelizantes

Compiladores paralelizantes caracterizam ambientes de paralelização automática, sendo responsáveis pela geração automática de programas paralelos a partir de sua versão seqüencial. A paralelização automática é normalmente a possibilidade mais simples de ser utilizada, uma vez que requer pouco ou até nenhum esforço extra do usuário, não sendo preciso nem modificar o programa, nem aprender uma nova linguagem. Entretanto, o desempenho obtido geralmente é modesto, a não ser em aplicações específicas onde o *speed-up* poder ser grande. Além disso, nem sempre esses compiladores estão disponíveis, principalmente para sistemas de memória distribuída. Normalmente exploram granularidade fina e apresentam baixa flexibilidade.

4.7.2 Extensões Paralelas

Extensões paralelas são bibliotecas que contêm um conjunto de instruções que complementam linguagens seqüenciais já existentes. Estes ambientes requerem mais trabalho do programador, mas ainda evitam a necessidade de aprendizagem de uma nova linguagem e a completa reescrita do código fonte (o usuário deve aprender uma extensão para uma linguagem já conhecida). Geralmente, o desempenho obtido é superior ao obtido pelos compiladores paralelizantes. Alguns dos exemplos de extensões paralelas são P4 (CARRIERO; GELERNTER, 1989), PVM (SUNDERAM et al., 1994), MS-MPI entre outros.

É importante salientar que normalmente esses ambientes utilizam a passagem de mensagens como mecanismo de comunicação, visto que formam uma arquitetura MIMD com memória distribuída. Entretanto, é possível que seja simulada uma memória compartilhada, ou ainda que o ambiente utilize memória compartilhada quando necessário de modo transparente à aplicação paralela.

4.7.3 Linguagens Concorrentes

O terceiro tipo de ferramenta está relacionado às linguagens criadas especialmente para o processamento paralelo, o que implica em tempo de aprendizagem de uma linguagem totalmente nova e restrita total do código fonte. Essas ferramentas tendem a fornecer melhor desempenho, de maneira a compensar a sobrecarga sobre o programador.

Outra vantagem destas linguagens é que geralmente elas possibilitam a construção de código bem estruturado, tornando fácil a identificação dos processos que estão executando em paralelo e a comunicação entre eles. Exemplos dessas linguagens são: C Paralelo, HPF e CPAR.

4.8 Conclusão

A escolha de uma ferramenta para programação paralela deve ser feita de acordo com os objetivos do programador. Compiladores paralelizáveis oferecem *speed-up* limitado, porém com sobrecarga nula sobre o programador. Por outro lado, linguagens concorrentes oferecem melhor desempenho, mas oferecem uma sobrecarga considerável sobre o programador. Num patamar intermediário de desempenho e sobrecarga do programador situam-se as extensões paralelas.

No próximo capítulo é apresentado o GPU, inicialmente vista exclusivamente como um dispositivo de processamento gráfico, mas que com o passar dos anos foi evoluindo para um dispositivo sólido e qualificado para o processamento maciço de dados em paralelo.

5 *Programação Genérica com GPU*

Como pode ser observado no Capítulo 4, a idéia básica por detrás do HPC é criar um ambiente onde muitos dados possam ser processados ao mesmo tempo, ou seja de forma paralela, como na Figura 5.1 onde tem-se uma entidade responsável pela organização da entrada dos dados, uma plataforma de processamento paralela e então uma ultima entidade responsável pela saída desses dados.

Se for generalizada a idéia de HPC dentro do modelo apresentado na Figura 5.1, pode-se ver o GPU também como uma das opções viáveis de dispositivo para este ambiente. Este capítulo é apresentado o GPU como um dispositivo de programação para a computação genérica, e descreve em detalhes o CUDA, uma arquitetura proposta pela empresa nVidia com o objetivo de facilitar o desenvolvimento destas aplicações de fim científico, ou seja, não apenas para o contexto de processamento gráfico tradicional.

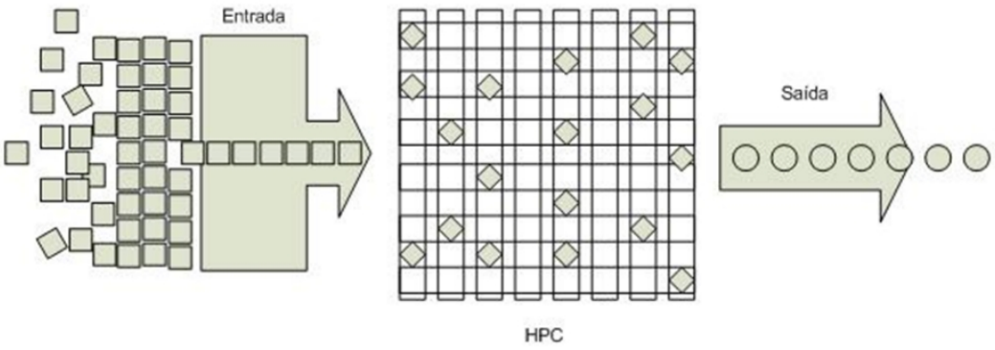

Figura 5.1: Modelo esquemático de processamento paralelo de dados

5.1 Evolução do GPU

O CPU é a parte de um computador que interpreta e vela as instruções do software. Na maioria dos CPU's essa tarefa é dividida entre uma unidade de controle que dirige o fluxo do programa e uma ou mais unidades de execução que executam operações em dados. Desde o seu surgimento, sempre teve em mente o aumento de seu poder computacional, mas sempre preservando a sua

principal característica, a sua flexibilidade para resolver problemas genéricos. Essa flexibilidade vem do fato de que o CPU segue as instruções de forma sequencial existentes em um *software*.

Como seu objetivo é o de atender as mais diversas necessidades computacionais, os CPU's também podem fazer o processamento gráfico, mas de forma modesta, pelas próprias exigências da área de processamento gráfico. Operações como *renderização*, aplicação de texturas e iluminação, ou desenho de sombras em cenas 3D, não são atividades triviais exigindo um grande poder computacional para a realização desses cálculos essencialmente matriciais. Desse cenário vê-se então a necessidade da criação de dispositivo que fosse especializado no processamento gráfico.

A idéia da criação de uma unidade de processamento de dados especializada para atividades gráficas, vem desde o começo dos anos 90. Antes da existência dos GPU, companhias como a *Silicon Graphics (SGI)* e *Evans & Sutherland* forneciam ao mercado equipamentos especializados em processamento gráfico intensivo, como o *Octave* da empresa *Silicon Graphics*. Estes sistemas gráficos introduziram muitos conceitos, entre eles a transformação de vértices e mapeamento de texturas, que são usados ainda hoje. Estes sistemas têm uma importância muito grande na história da computação gráfica, mas por causa do alto custo dessas estações de trabalho e do sucesso dos GPU's, essas estações de trabalho são coisa do passado.

Pode-se organizar os GPU's em gerações sendo estas organizadas em seis gerações (COMBA et al., 2003):

- Primeira
  - Período:Até 1998
  - Representantes
    - nVidia TNT2
    - ATI Rag
  - Características relevantes
    - Interpolação: capacidade para calcular os pixels a partir dos vértices de um triângulo e aplicar texturas
- Segunda
  - Período: 1999 a 2000
  - Representantes
    - GeForce 256,
    - GF2 Radeon 7500

  - Características relevantes
    - Translação de Vértices
    - Cálculo de Iluminação (por vértice)

- Terceira
  - Período: 2001
  - Representantes
    - GeForce 3 e 4,
    - Radeon 8500
  - Características relevantes
    - Programação ao nível dos vértices

- Quarta
  - Período: 2002 a 2003
  - Representantes
    - GeForce FX,
    - ATI Radeon 9700
  - Características relevantes
    - Programação ao nível do pixel

- Quinta
  - Período: 2004 a 2005
  - Representantes
    - nVidia 6800, 7800,
    - ATI X1800
  - Características relevantes
    - Acesso a memória pelo programa (vertex)

- Sexta
  - Período: Começa em 2006
  - Representante: nVidia GeForce 8800
  - Características relevantes

∗ Apresentação da arquitetura CUDA

O grande marco do uso do GPU como um dispositivo de programação geral deu-se na terceira geração, no ano de 2001, quando os processadores do tipo *Vertex* começaram a poder ser programados pelo desenvolvedor externo, ou seja, era possível desenvolver uma aplicação (módulo) que iria substituir as funcionalidades naturais exercidas pelo módulo de *Vertex* da GPU.

Essa tendência de dispositivo programável ganhou mais robustez pelas próximas gerações; na quarta, com a capacidade de desenvolver módulos executados pelos processadores do tipo *Fragment*, e na quinta, onde os módulos executados no *Vertex* podem ter acesso à memória. Um *pipeline* com esse módulo de programação pode ser visto na Figura 5.2.

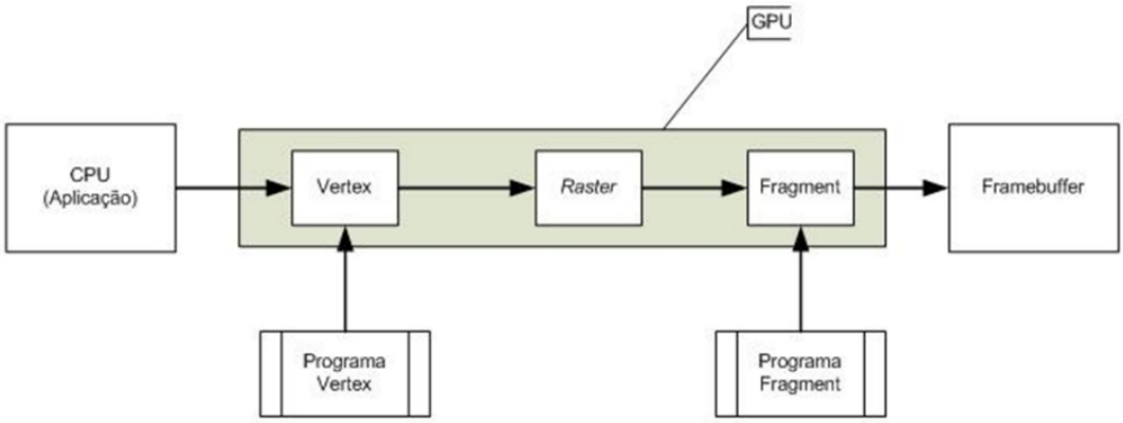

Figura 5.2: Pipeline do desenvolvimento tradicional em GPU

5.2 GPU como Dispositivo de Processamento Genérico

Nos últimos 10 anos tem-se observado a evolução dos GPU's até então um hardware especializado no processamento gráfico e saída de vídeo, para um dispositivo de processamento maciço e paralelo de dados para computação geral. O poder de processamento de dados dos GPU's cresceu muito mais rápido que o dos CPU's, observado no gráfico da Figura 5.3, onde tem-se uma visão comparativa do crescimento GPU x CPU tomando como base a capacidade de realizar operações de ponto flutuante.

A principal razão para este crescimento acelerado dos GPU's com relação aos CPU's deve-se ao fato de que os GPU's já nasceram com o foco de sua necessidade em computação intensiva, com relação a processamento maciço de dados e computação em paralelo, justamente características mínimas necessárias para atender as necessidades do cenário da computação gráfica, como renderização de imagens, sombras em cenas 3D entre outras. Desta forma o projeto do GPU leva

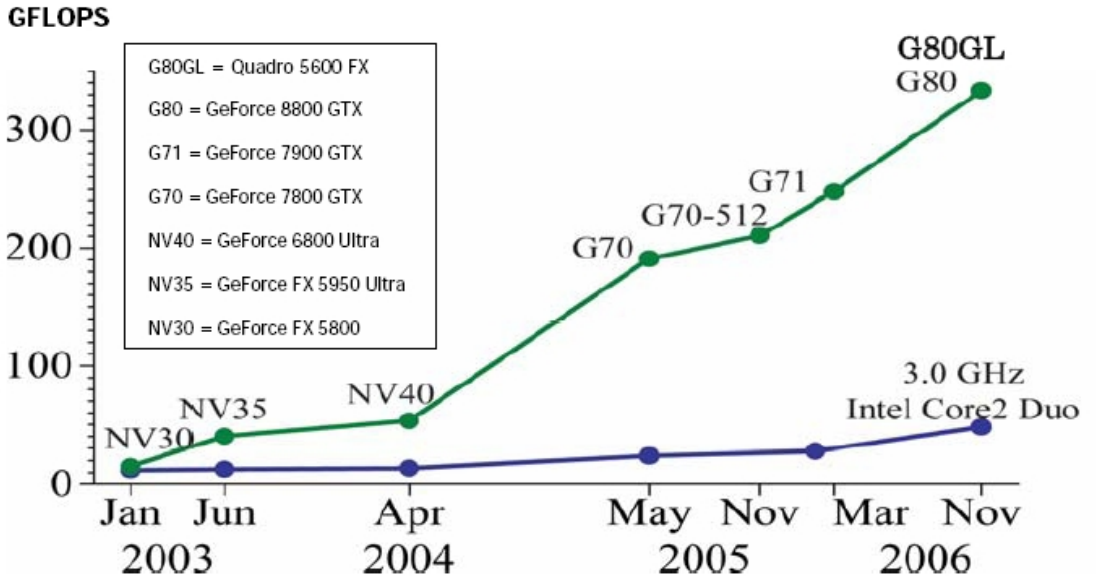

Figura 5.3: Operações de ponto flutuante por segundo (CPU x GPU)

em consideração a existência de mais transistores dedicados a um melhor processamento e controle de fluxo de dados, como ilustrado esquematicamente na Figura 5.4, onde são mostrados os principais elementos: ALU, cache, controle e DRAM para um CPU (Figura 5.4a) e para um GPU (Figura 5.4b).

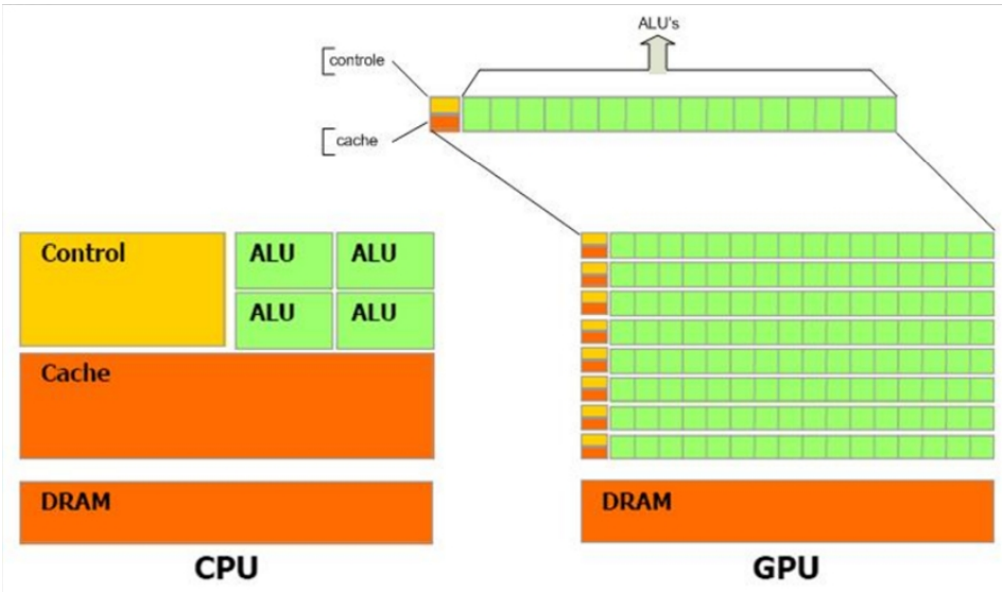

Figura 5.4: GPU Destina mais Transistores para o Processamento de Dados

Um GPU pode ser visto como sendo dispositivo especializado em endereçar problemas que podem ser expressos por meio de um modelo de computação paralelo. O mesmo programa, com intensidade aritmética elevada, é executado em muitos dados de forma paralela. Porque o mesmo programa é executado para cada elemento de dados, não existe uma exigência de um controle de

fluxo de dados sofisticado; e porque é executada em muitos elementos de dados e tem a intensidade aritmética elevada, a latência do acesso de memória pode ser "escondida"por meio de cálculos em vez de grandes *caches* de dados.

Muitas aplicações que processam séries de dados grandes organizadas de forma matricial/vetorial podem usar um modelo de computação paralelo. Em processos de *renderizacão* 3D grandes arranjos de *pixels* e vértices são organizados de forma que possam ser processados de forma paralela usando *threads*. Da mesma forma, aplicações de processamento de imagens, codificação e decodificação de vídeo, *scaling*, visão estéreo, redes neurais artificiais e reconhecimento de padrões podem ser processadas em blocos de dados e *pixels* por meio de *threads* paralelos. De fato, muitos algoritmos, mesmo que fora da área de processamento de imagens, podem ser acelerados por meio da paralelização do processamento de dados, principalmente processamento de sinais, simulação de efeitos físicos, computação de modelos financeiros ou biológicos.

Assumindo a idéia de que um HPC resume-se a um conjunto de dados sendo processados de forma maciçamente paralela, pode-se encontrar uma similaridade dentro do *pipeline* do GPU dentro deste contexto, conforme pode ser observado na Figura 5.5. Basicamente pode-se resumir o *pipeline* do GPU em três blocos de processamento de dados sendo o *Vertex Shader*, o *Fragment Shader* e o processo de *Raster*.

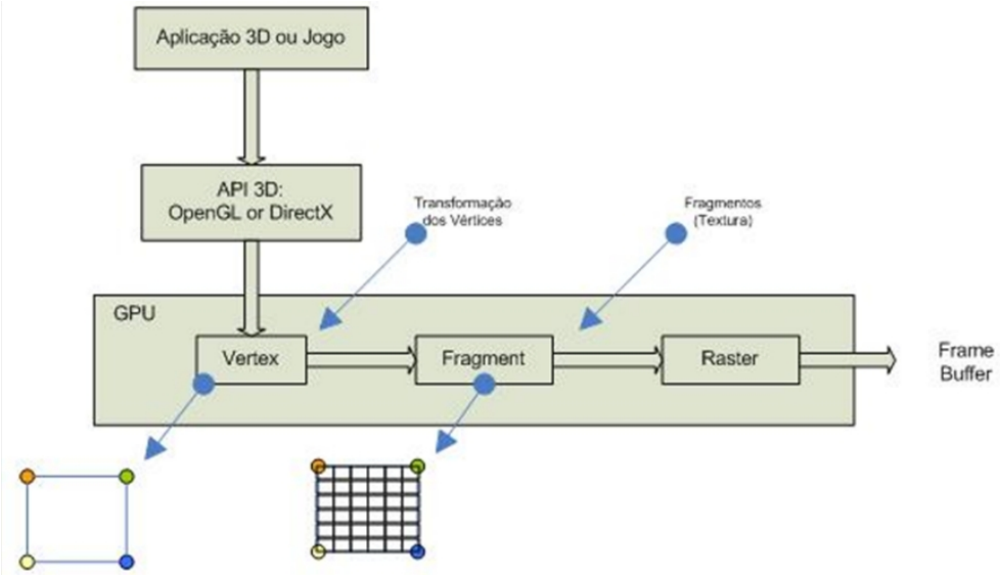

Figura 5.5: *Pipeline* tradicional da placa gráfica

A operação do *Vertex Shader* é o primeiro estágio do processamento dentro do *pipeline*, nele a aplicação gráfica informa a quantidade de vértices e como estes se relacionam na formação de figuras geométricas, conforme pode ser observado na Figura 5.5, a operação de *Vertex Shader* recebe 4 vértices cujas coordenadas montam a figura de um quadrado.

O próximo estágio do *pipeline* é o *Fragment Shader*, que recebe do *Vertex Shader* não apenas os vértices de um quadrado, mas sim um quadrado que tem em seu interior preenchido por uma textura, que pode ser observada como sendo cada posição dessa textura por um *pixel*, ou seja, uma matriz de *pixel*. É o *Fragment Shader* o responsável pelas operações de manipulação dos valores desses pixels. Por fim a figura, que no exemplo apresentado da Figura 5.5 é um quadrado, sai do *Fragment Shader* como um quadrado com uma textura já processada que passa para o ultimo estágio do *pipeline* que é justamente o *Raster* dessa imagem que o coloca no *Frame Buffer* para então ser apresentado na tela.

Uma vez comparado os dois *pipelines*, do HPC e do GPU, pode-se ver uma similaridade entre os processos que é apresentado na Tabela 5.1.

HPC	GPU
Entrada de Dados	Vertex Shader
Grid de Processamento Paralelo	Fragment Shader
Saída de Dados	Raster

Tabela 5.1: Comparação entre HPC x GPU

Pode-se observar uma evolução significativa do dispositivo de processamento gráfico no qual o hardware gráfico apresenta uma mudança na implementação de seu *pipeline*, passando de um conjunto de funções, para programas especiais implementados por um desenvolvedor que podem ser executados para cada vértice ou fragmento. Esta capacidade, de ser programável permite a implementação de diversos algoritmos diretamente no *hardware* gráfico.

Entretanto, apesar de todo o poder de computação paralelo e benefícios que poderiam ser alcançados por meio do uso dos GPU's, ainda permanece complicado sua utilização em aplicações não gráficas podendo-se destacar:

- Para desenvolver uma aplicação que é executada no GPU, esta só poderia ser executada por meio de comandos gráficos existentes em um API especializado. Isso impõe uma curva de aprendizagem muito elevada para iniciantes, principalmente se estes não estiverem acostumados com processamento gráfico;
- Em muitos casos esses API's, obrigatórios, para o desenvolvimento de aplicações para o GPU, são tão especializados no contexto de processamento gráfico que dificultam e/ou inviabilizam o desenvolvimento de aplicação não gráfica;
- A memória de um GPU (DRAM) pode ser lida em qualquer momento, mas escrita apenas em um momento do *pipeline*. Com isso as aplicações perdem a flexibilidade e

- Algumas aplicações, quando modeladas de forma inadequada, são capazes de exceder a largura de faixa da memória do GPU (DRAM), subutilizando seu poder computacional.

Como pode-se perceber, o GPU, apesar de ser um dispositivo, já qualificado, para o processamento maciço de dados segundo uma arquitetura de computação paralela, ainda apresentava uma série de problemas/dificuldades quanto ao desenvolvimento de aplicações fora do contexto gráfico. Visando suprir essa deficiência foi desenvolvimento uma nova arquitetura para o GPU chamada de CUDA (*Compute Unified Device Architecture*) que visa atender as necessidades quanto ao desenvolvimento de aplicação fora do contexto gráfico.

5.3 CUDA

O desenvolvimento de aplicações que usam o GPU como um dispositivo de processamento de dados paralelo "não convencional", ou seja, não para o processamento gráfico especificamente como renderização de imagens, vem aumentando. Contudo o uso de um GPU como um dispositivo desse tipo exige uma adaptação do *pipeline* tradicional da placa gráfica, forçando que o desenvolvedor assuma a responsabilidade por certos pontos de controle dentro desse processamento, isso por meio de bibliotecas gráficas que possuem um API para GPU's programáveis.

CUDA é uma nova arquitetura de *hardware* e *software* que foi desenvolvido com o objetivo central de gerenciar o processamento de dados paralelo dentro do dispositivo GPU sem que haja a necessidade de fazer o mapeamento das rotinas e assumir a responsabilidade por momentos da execução do *pipeline* gráfico, por meio de API gráfico.

Na Figura 5.6 tem-se a pilha de softwares do ambiente do CUDA , sendo esta pilha composta, não obrigatoriamente, por 4 camadas de software sendo estas (a) Aplicação, que é o software implementado pelo pesquisador que faz uso do GPU como dispositivo de processamento de dados, (b) CUDA Library que é um conjunto de bibliotecas matemáticas, como o CUBLAS, uma extensão do BLAS uma biblioteca de funções de álgebra implementadas em FORTRAN e CUFFT uma biblioteca de transformada rápida de Fourier de 1, 2 e 3 dimensões. (c) CUDA Runtime onde as rotinas de outras bibliotecas gráficas como OpenGL e DirectX fazem acesso a processamento no GPU e o (d) CUDA Driver que é o API de comunicação direta com o GPU.

Visando facilitar o processo de desenvolvimento de soluções computacionais de fins genéricos, não apenas gráfico o CUDA propicia ao GPU acesso direto a memória tanto para escrita (Figura 5.7) quanto para leitura (Figura 5.8), da mesma forma como funciona um CPU convencional.

Nessas Figuras (Figura 5.7 e Figura 5.8) os dados d_i são lidos ou escritos na memória pelos

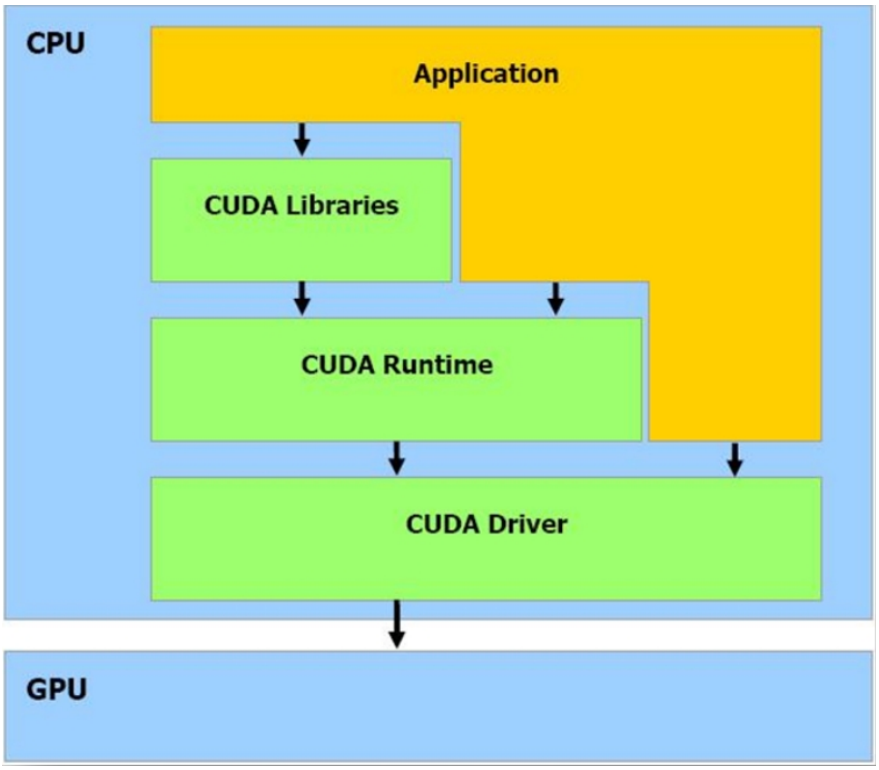

Figura 5.6: Pilha de Software CUDA

ALU‘s.

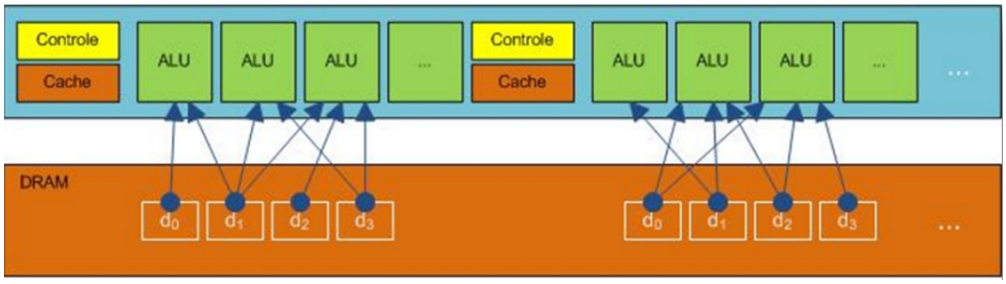

Figura 5.7: GPU Acessando a memoória para leitura

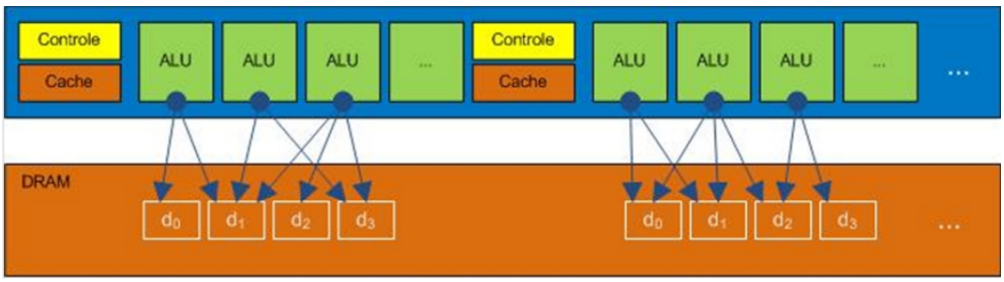

Figura 5.8: GPU Acessando a memória para escrita

Nesta arquitetura existe um cache de dados paralelo, e uma memória compartilhada, que possui uma alta velocidade tanto para acesso a escrita quanto para a leitura, como pode ser observado na

Figura 5.9. As aplicações se beneficiam desta estrutura por minimizar *overfetch* e *round-trips* da DRAM além de reduzir a necessidade/dependência da largura de banda de acesso a DRAM.

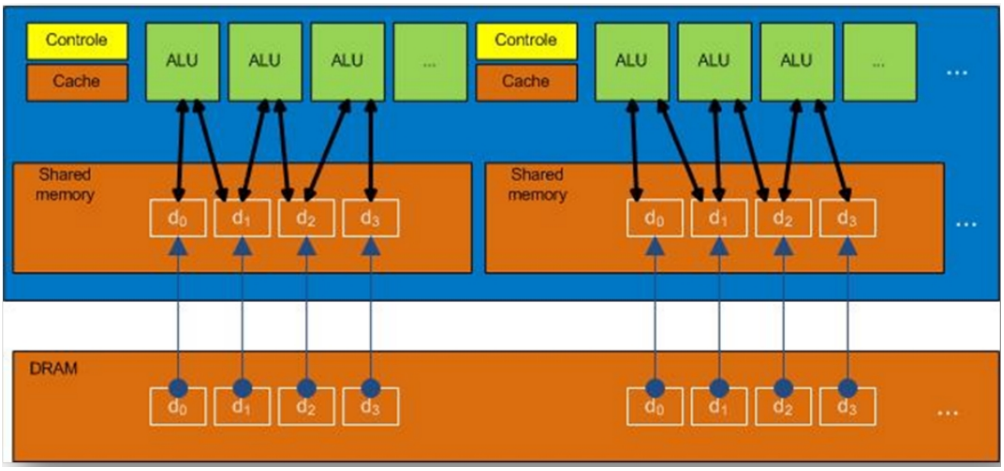

Figura 5.9: Memória compartilhada entre as ALUs de um bloco

5.3.1 Modelo de Programação

No desenvolvimento de uma aplicação paralela por meio do CUDA o GPU é visto como um dispositivo do computador capaz de executar um grande número de *threads* em paralelo. O GPU funciona como um coprocessador do CPU, que no contexto do CUDA recebe o nome de *host*.

A parte da aplicação mais indicada para ser processada no dispositivo é uma função executada várias vezes com diferentes dados. Estas funções devem ser isoladas e implementadas dentro da extensão do CUDA e são chamadas de *kernel* por serem executadas dentro do dispositivo.

Ambos *host* e dispositivo (GPU) possuem uma memória DRAM uma chamada de memória do dispositivo e outra memória do *host*. Antes que seja feita a chamada de um *kernel* deve-se fazer a transferência desses dados entre as duas memórias. O CUDA disponibiliza um conjunto de funções para esta funcionalidade (movimentação de dados entre os dois tipos de memória).

Processamento de *threads* em Lote

Quando uma aplicação de *host* faz uma chamada a um *kernel* esta é executada por um conjunto de *threads* organizados em blocos de execução. Esses blocos por sua vez, estão agrupados em grade de blocos.

Um bloco de *threads* é um lote de *threads* que trabalham em conjunto de forma cooperativa para ter uma melhor eficiência do uso dos dados e memórias compartilhadas e tem seus processamentos sincronizados. Cada *thread* dentro de um bloco é identificado por um *threadID* que é uma

combinação do número desse *thread* com o bloco no qual está inserido.

A formação de um valor de uma *threadID* é complexa e para auxiliar nesse processo pode-se especificar um bloco como tendo duas ou três dimensões de tamanho arbitrário e identificar cada *thread* usando um índice composto de duas ou três instâncias, como apresentado na Tabela 5.2, onde D_x, D_y, D_z são dimensões dos blocos; x, y, z são as coordenadas; e *threadID* são obtidos calculando as expressõs apresentadas.

Dimensão do Bloco	Coordenada da Thread	threadID
D_x, D_y	x, y	$x + yD_x$
D_x, D_y, D_z	x, y, z	$x + yD_x + zD_xD_y$

Tabela 5.2: Formação do endereço da thread dentro do bloco

O número de *threads* que um bloco pode conter é limitado. Como já dito, blocos com a mesma dimensionalidade que trabalhem na execução de um mesmo *kernel* podem ser agrupados dentro de uma grade de blocos de *threads*. A chamada desse *kernel* é realizada por meio de uma sintaxe específica onde são informados além dos parâmetros normais da função a ser processada no dispositivo, dados referentes Ãă grade (Dg), bloco (Db) e memória a ser alocada (Ns) como é mostrado na chamada a seguir:

FuncKernel«<Dg, Db, Ns»>(Parametros)

Assim como os *threads* os blocos também possuem um número de identificação dentro de uma grade, seguindo uma regra similar da formação do endereço dos *threads*, como apresentado na Tabela 5.3, onde D_x, D_y indicam as dimensões da grade, x, y as coordernadas dos blocos e *blockID* o número de identificação do bloco calculado pela expressão apresentada.

Dimensão do Grid	Coordenada do Bloco	blockID
D_x, D_y	x, y	$x + yD_x$

Tabela 5.3: Formação do endereço do blockID dentro de um gride

Na Figura 5.10 é apresentado um esquema da estrutura de execução das *threads* dentro do dispositivo. Nesta pode ser observada a separação entre os dois *hardwares host* (CPU) e o dispositivo (GPU) onde os *kernels* chamados pela aplicação no *host* são enviados para o dispositivo, onde os processamentos de threads organizados em blocos são distribuídos em grades de processamento.

Vale chamar atenção aqui ao fato de que para *kernels* distintos existem configurações de grade e blocos distintos quanto a sua dimensionalidade, como pode ser observado na Figura 5.10 onde

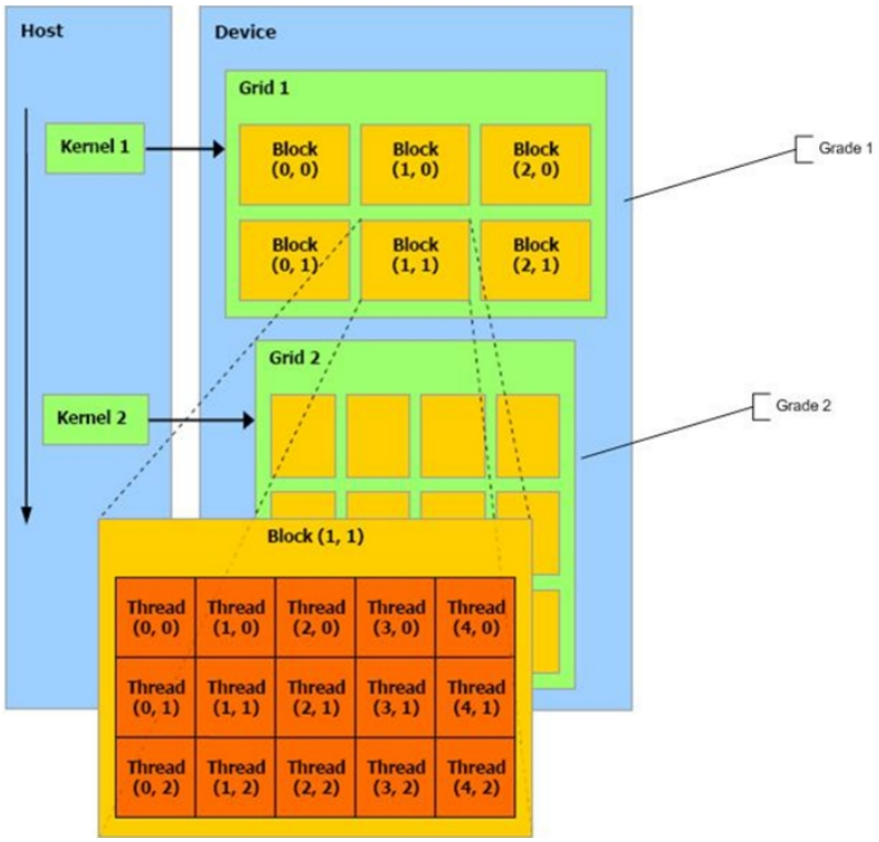

Figura 5.10: Endereço das threads dentro dos blocos.interno às grades (*grid* em inglês)

o tamanho dos blocos e do grade usado no processamento do kernel 2 é diferente do usado pelo *kernel* 1.

Modelo de Memória

O *thread* é executado dentro do dispositivo e possui apenas acesso à memória (DRAM) existente dentro deste, segundo um conjunto de regras de acesso apresentado na Figura 5.11 e detalhado na Tabela 5.4.

Os *threads* podem fazer acesso aos registradores (*register*) e mamória local para leitora e escrita. A memória compartilhada (*shared*) é acessadas pelos blocos para escrita e leitura. A memória global é acessada pela grade para leitura e escrita. As memórias de *constant* e texturas são acessadas pela grade somente para leitura.

Os espaços globais, constantes, e de textura podem ser lidos ou escritos pelo *host* e são persistentes através das chamadas da *kernel* pela mesma aplicação.

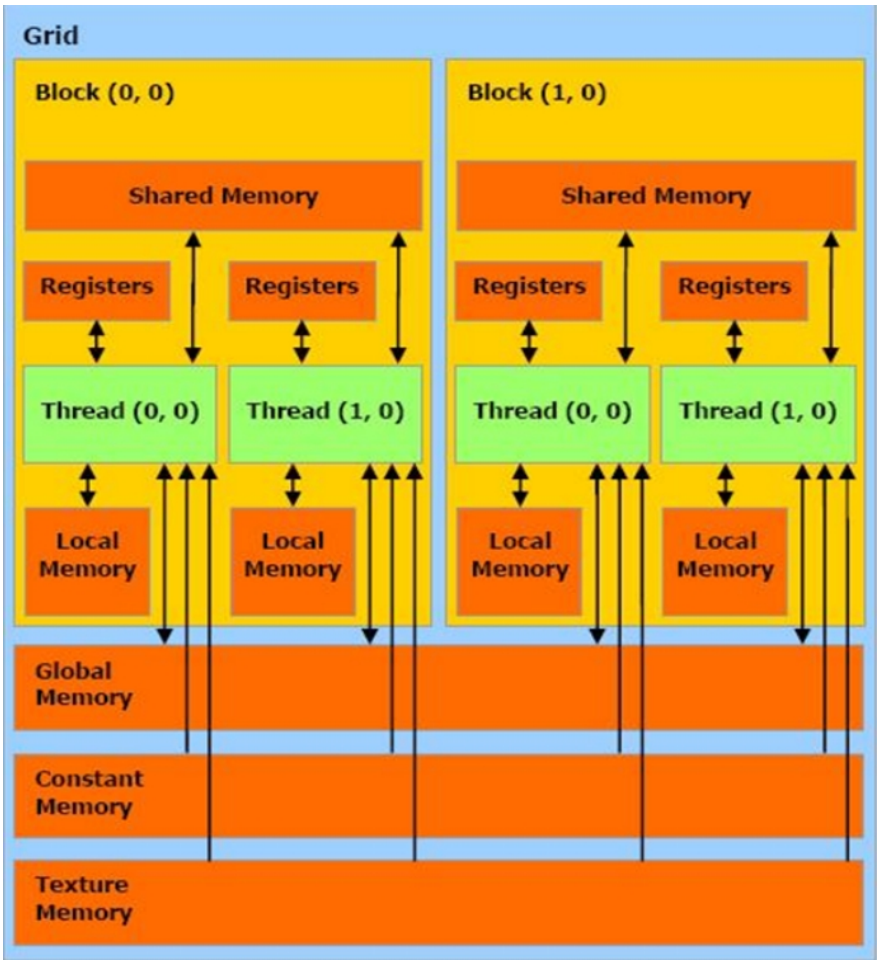

Figura 5.11: Modelo de Memória no CUDA

Espaço de Memória	Quando acessado por:	Regra de Acesso
Register	Pela *Thread*	Leitura/Escrita
Local	Pela *Thread*	Leitura/Escrita
Shared	Pelo Bloco	Leitura/Escrita
Global	Pela Grade	Leitura/Escrita
Constant	Pela Grade	Só leitura
Texture	Pela Grade	Só leitura

Tabela 5.4: Politicas de acesso a memória da GPU

Implementação em Hardware

O dispositivo é uma implementação de um conjunto de multiprocessadores, como apresentado na Figura 5.12, onde cada multiprocessador é um SIMD, onde a cada *clock* cada processador executa a mesma instrução em um conjunto diferente de dados.

Cada multiprocessador tem um espaço de memória com uma das quatro funções:

- Um conjunto de registradores locais de 32-bits por processador;

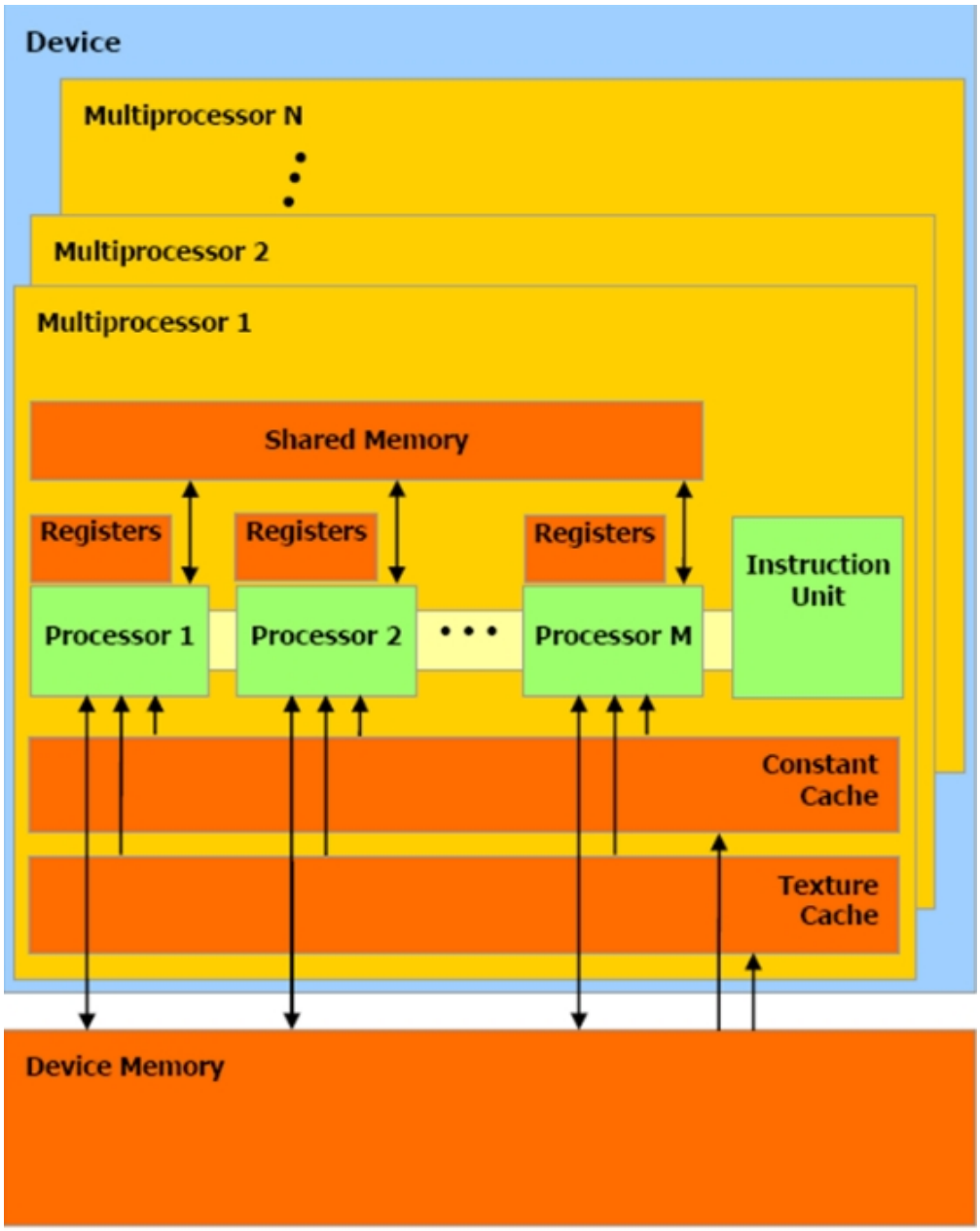

Figura 5.12: Modelo do Hardware - Processamento SIMD de Dados

- Um cache de dados paralelos ou uma memória compartilhada;
- Um cache Constant do tipo apenas de leitura que é compartilhado entre todos os processadores ou
- Um cache de Textura apenas de leitura também compartilhado entre todos os processadores.

Modelo de Execução

Uma grade de blocos de *threads* é executada no dispositivo executando um ou mais blocos em cada multiprocessador usando tempo compartilhado. Cada bloco é dividido em grupos de *threads*

chamado *warps*; cada um destes *warps* contem o mesmo número de *threads*, que é chamado de *warp size*, e é executado pelo multiprocessador em uma organização SIMD; um escalador (*scheduler*) de *threads* periodicamente comuta um *warp* para outro com o objetivo de maximizar o uso dos recursos computacionais do multiprocessador. O modo como um bloco é dividido em *warps* é sempre o mesmo; cada *warp* contém *threads* de identificação sucessivas, crescentes com o primeiro *warp* começando no *thread* 0.

Um bloco é processado em um único multiprocessador, de forma que o espaço de memória compartilhado do dispositivo conduz a acessos muito rápidos. Os registradores do multiprocessador são alocados entre os *threads* do bloco. Se o número de registradores alocados por *threads* multiplicados pelo número de *threads* no bloco é maior que o número total de registradores por multiprocessador, o bloco não pode ser executado e o CPU será sinalizado.

Vários blocos podem ser processados simultaneamente pelo mesmo multiprocessador alocando os registradores do multiprocessor e memória compartilhada entre os blocos. A ordem de emissão dos *warps* dentro de um bloco é indefinida, mas a sua execução pode ser sincronizada.

A ordem de emissão dos blocos dentro de uma grade é indefinida e não há nenhum mecanismo de sincronização entre blocos, assim *threads* de dois blocos diferentes da mesma grade não podem comunicar seguramente entre si por meio da memória global durante a execução da grade.

Se uma instrução não-atômica executada por um *warp* escreve na mesma posição de memória global ou compartilhada para mais que um *thread* de um *warp*, o número de escritas seriais naquela posição e a ordem em que elas acontecem são indefinidos, mas é garantido sucesso para um deles. Se uma instrução atômica executada por um *warp* lê, modifica, e escreve na mesma posição de memória global para mais que um dos *threads*, cada leitura, modificação, e escrita àquela posição acontece, mas a sua ordem é indefinida.

API de Desenvolvimento

A principal meta do API do CUDA é disponibilizar um mecanismo familiar e simples para os programadores, para isso usa uma extensão da linguagem C facilitando assim o desenvolvimento de aplicativos que usem o GPU como dispositivo de processamento de maciço de dados paralelo. Basicamente este API consiste de:

- Um mínimo de extensões na linguagem C ANSI
- Uma biblioteca em tempo de execução organizada em:

  Um conjunto de componentes que são executados no host e disponibiliza funções de

controle de acesso a um ou mais dispositivos por meio deste *host*;

Componentes do dispositivo que são executados no dispositivo e representam um conjunto de funções específicas deste e

Componentes comuns tanto ao dispositivo quando ao *host*.

Extensões da Linguagem C

Pode-se organizar as extensões de linguagem providas em quatro categorias principais:

- Qualificadores de tipos de funções;
- Qualificadores de tipos de variáveis;
- Novas diretivas para especificar como o *kernel* é executado no dispositivo a partir do *host* e
- Quatro variáveis embutidas para especificar as dimensões da grade, blocos e *threadsID*.

Estas categorias serão descritas em mais detalhes a seguir nos próximas seções.

Qualificadores de Funções

Os qualificadores de função especificam onde os mesmas são executadas e podem ser: *device*,*global* e *host*, apresentados na Tabela 5.7:

Qualificador	Notas de Uso
__device__	É executado no dispositivo e pode ser chamada apenas por funções dentro do dispositivo
__global__	É executada no dispositivo e pode ser chamada apenas por funções no *host*
__host__	É executada no *host* e pode ser chamada apenas por funções no *host*

Tabela 5.5: Qualificadores de funções CUDA

Qualificadores de Variáveis

Os qualificadores de variáveis especificam em que espaço de memórias as mesmas serão alocadas, e são apresentadas na Tabela 5.6.

Variáveis Embutidas

As variáveis embutidas são descritores do ambiente CUDA, e são descritas na Tabela 5.7.

Qualificador	Notas de Uso
__device__	Reside na memória global, Tem seu tempo de vida ligada a aplicação É acessível por todas as threads de um grade e pelo *host* por meio das funções da biblioteca (*runtime*)
__constant__	Reside no espaço de memória *Constant*, Tem seu tempo de vida ligado ao da aplicação É acessível por todas as threads pertencentes a uma grade e pelo *host* por meio das funções da biblioteca (*runtime*).
__shared__	Reside no espaço de memória *shared* para um bloco de threads Tem seu tempo de vida ligado ao tempo de vida de um bloco É acessível apenas a todas *threads* de um bloco

Tabela 5.6: Qualificadores de variáveis CUDA

Variável	Descrição
gridDim	Tipo dim3, Contem das dimensões do grade
blockIdx	Tipo uint3, Contém o índice do bloco pertencente a uma grade
blockDim	Tipo dim3, Contem a dimensão do bloco
threadIdx	Tipo uint3, Contem o índice do thread pertencente a um bloco

Tabela 5.7: Variáveis embutidas CUDA

NVCC

O código a ser executado no dispositivo (*kernel*) deve ser compilado por meio do nvcc, um compilador de linha de comando que é utilizado para esta finalidade, que apresenta uma interface simples e familiar aos usuários de outros compiladores por linha de comando como o gcc.

Basicamente o fluxo de trabalho do nvcc consiste em separar o código que é executado no *host* e compilar o código a ser executado no dispositivo. Este código objeto recebe o nome de *cubin*. O código do *host*, separado na fase anterior é executado por outro compilador.

Na execução das aplicações os *cubins* são passados para o *driver* do CUDA e executados pelo dispositivo.

5.4 Conclusão

Os GPUs deixaram de ser dispositivos especializados para processamento gráfico e se tornaram uma opção viável para o processamento de aplicativos gerais que necessitam de um processamento maciço de dados de forma paralela.

No próximo capítulo é descrito o trabalho realizado usando a rede Neocognitron para o reco-

nhecimento facial.

6 *Trabalho Desenvolvido*

O reconhecimento facial não é um processo trivial de reconhecimento de padrões, não pode ser considerada a face humana como um simples mapa de padrões a ser classificado, uma vez que esta mesma face está sujeita a mudanças, de ordem natural, como as ações do tempo, do envelhecimento, de ornamentos, como brincos, chapéus e outros adereços e mesmo artificiais ou provocadas, como as cirurgias plásticas e acidentes (cicatrizes). É necessário, portanto entender o processo de reconhecimento facial dentro da própria natureza inquieta e evolutiva do ser humano.

Neste capítulo é apresentado o desenvolvimento do trabalho da dissertação de mestrado; o reconhecimento facial por meio da rede neural artificial Neocognitron tendo seu processamento dentro da arquitetura CUDA. Como este ambiente é naturalmente de processamento paralelo, espera-se um aumento da eficácia, com a manutenção da acuracidade da rede juntamente com a redução do seu custo computacional. Nas próximas seções é apresentado em detalhes o trabalho realizado.

6.1 Trabalhos Anteriores

Em (RIBEIRO, 2002), é apresentado um estudo da Paralelização da Rede Neural Neocognitron, para uma aplicação de reconhecimento facial e minutiae em Impressões Digitais, usando uma arquitetura de cluster de multiprocessadores SMPs interligados nos nós de processamento por meio de um sistema de comunicação *Fast-Ethernet* - 100Mbps InfoServer 3030 - Itautec. Neste trabalho foi utilizado MPI como interface de passagem de mensagens entre os processadores.

Seus resultados mostram que quanto maior a granularidade da paralelização, melhor é o desempenho do processamento, citando como exemplo, o melhor *speed-up* no reconhecimento de impressões digitais em relação ao reconhecimento facial. Seus resultados obtidos fornecem subsídios para o dimensionamento da rede Neocognitron, aumentando ou diminuindo o número de estágios e o tamanho dos planos celulares da rede neural e da arquitetura, variando a quantidade de nós de processamento e a velocidade de comunicação entre os processadores.

6.2 Processamento da Rede Neocognitron

O objetivo principal do trabalho é realizar o reconhecimento facial por meio da rede Neocognitron, sendo esta processada dentro do GPU/CUDA. Para este projeto foi utilizada uma rede Neocognitron com 3 estágios não incluindo a camada de entrada. Na Figura 6.1, pode ser observado o diagrama de blocos do sistema de reconhecimento facial usando a rede Neocognitron dentro do contexto.

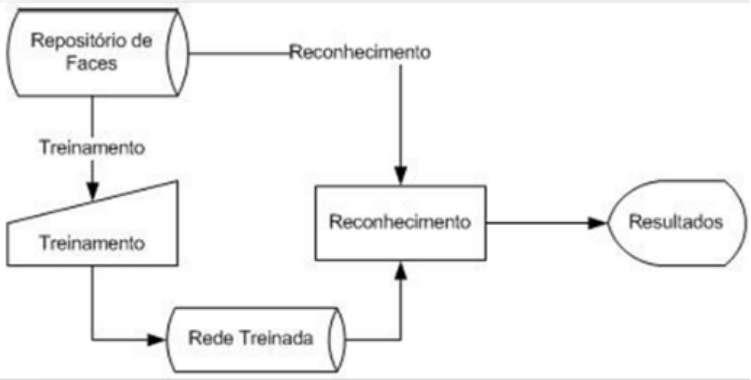

Figura 6.1: Diagrama de Blocos do Sistema de Reconhecimento Facial da rede Neocognitron dentro do projeto

Dentro de um sistema de reconhecimento facial, pode-se dividir este em dois momentos: o treinamento e a reconhecimento. Este projeto tem foco no "Reconhecimento"sendo a fase de Treinamento realizada por outro projeto, implementado em Delphi. Ambos os projetos, contam com o mesmo *Repositório de Faces*. O projeto responsável pelo *Treinamento* gera um repositório de dados, chamado de *Rede Treinada*, utilizada para o processo de *Reconhecimento*.

Um "Repositório de Faces"é um conjunto de arquivos de imagens onde cada imagem é uma face a ser utilizada na rede, para o treinamento e/ou reconhecimento. Foram utilizados dois modelos de repositórios de faces, um criado pelo Marcelo Hiroshi Hirakuri em sua dissertação de mestrado na Universidade Federal de São Carlos (HIRAKURI, 2003) e outra CMU-PIE, conforme podem ser observadas nas Figura 6.2 e Figura 6.3, respectivamente.

Figura 6.2: Repositório de Faces desenvolvido na UFSCar

A base de dados CMU PIE (SIM; BAKER; BSAT, 2003) consiste de um grande número de imagens de pessoas, com diferentes poses, iluminações, e expressões faciais. Foram usadas 13

Figura 6.3: Repositório de Fases CMU-PIE

câmeras, 9 na mesma linha horizontal, cada uma separada de 22.5. Outras 4 câmeras incluem 2 acima e abaixo da câmera central, e 2 nos cantos da sala. Na Figura 47, as diferentes posições de câmeras são identificadas pelas notações $c02m...c34$. Para a obtenção da variação de iluminação, é usado um sistema com 21 *flashes*. Capturando imagens com e sem a luz de fundo, são obtidas 43 diferentes condições de iluminação. Para a obtenção de uma variedade de expressões faciais foi solicitado para as pessoas as expressões neutras, sorriso, piscando, e falando. O banco de dados consiste de 41368 imagens de 68 pessoas.

Foram escolhidas aleatoriamente 10 pessoas da base de dados CMU PIE (4002, 4014, 4036, 4047, 4048, 4052, 4057,4062, 4063, e 4067) (SIM; BAKER; BSAT, 2003) Como este trabalho consiste no reconhecimento, foram selecionadas as imagens de pessoas falando, devido a existência de 60 imagens por pose, por pessoa. Assim, durante os experimentos, foram usadas as imagens frontais das pessoas (640x486), após a captação da parte facial e redução para o tamanho (57x57), como mostrado na Figura 6.4.

Figura 6.4: Amostra de uma imagem da base de dados CMU PIE, e obtenção de uma imagem facial usando captura da face e redução

Este Repositório de Faces fornece um conjunto de faces que são utilizadas para a fase de treinamento da rede. O processo de treinamento da Neocognitron foi descrito no capítulo na seção Treinamento da Rede.

A fase de treinamento é realizada por um aplicativo desenvolvido em Delphi (SAITO et al., 2005) e que tem como produto de sua atividade a geração de um repositório de dados. Este é

formado por um conjunto de três tipos de arquivos binários: um para armazenar o número de planos de cada estágio, outro tipo para armazenar os $pesos - a$ e $pesos - b$ treinados pela rede e um terceiro com a saída utilizada pela camada complexa da rede de cada estágio.

Na Figura 6.5, pode-se ver o processamento da rede Neocognitron, referente ao processo de reconhecimento, objeto de estudo deste trabalho. Nele, pode ser observado os três estágios da rede, representados por: estágio 1, formado pelo conjunto das camadas US1, UC1 e UV1; estágio 2, formado pelo conjunto das camadas US2, UC2 e UV2 e o estágio 3, formado pelo conjunto das camadas US3, UC3 e UV3. Também é apresentada a camada de entrada, identificada por U0.

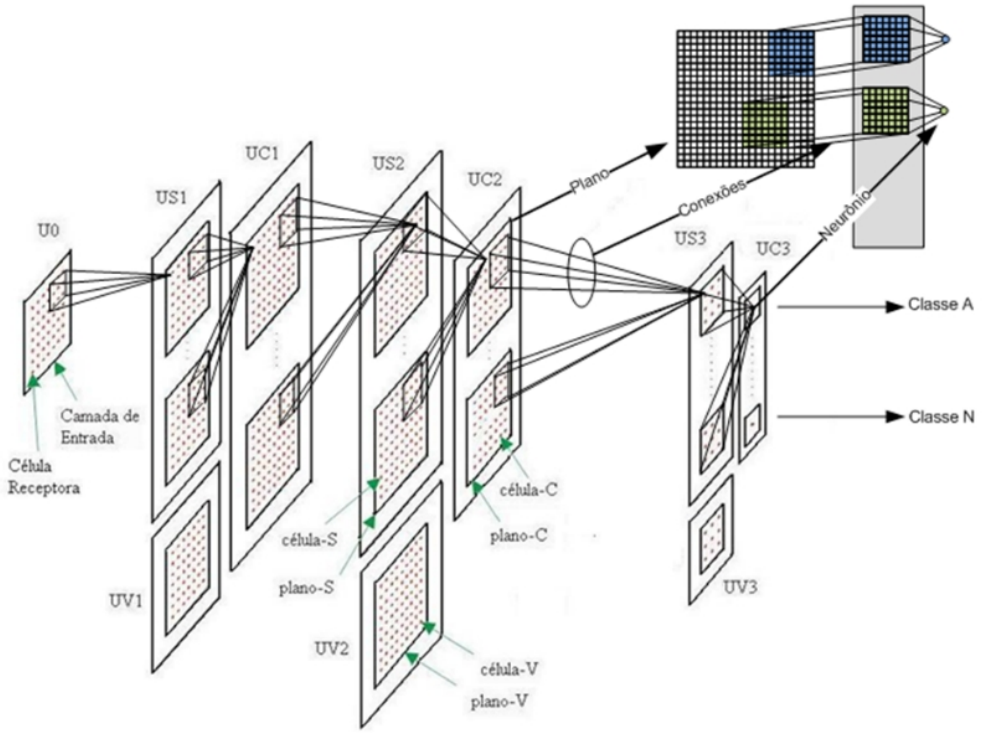

Figura 6.5: Processamento da Rede Neocognitron

Na Tabela 6.1 é apresentada a dimensionalidade dos pesos utilizados na rede, segundo o estágio onde este é aplicado. Os pesos a e b, como vistos no capítulo 3, são obtidos pelo processo de treinamento da rede, que no escopo deste projeto é realizado pela aplicação Delphi.

Estágio	Peso-a	Peso-b	Peso-c	Peso-d
1	7x7	1x1	7x7	5x5
2	7x7	1x1	7x7	5x5
3	5x5	1x1	5x5	3x3

Tabela 6.1: Dimensionalidade dos pesos utilizados na rede

Os pesos c e d são fixos e definidos no momento da implementação da rede. Nas Tabela 6.2, 6.3 e 6.4 são apresentadas as matrizes do $peso - c$ utilizadas nos estágios 1, 2 e 3 respectivamente.

0.017361	0.019231	0.019231	0.019231	0.019231	0.019231	0.017361
0.019231	0.019231	0.021635	0.021635	0.021635	0.019231	0.019231
0.019231	0.021635	0.021635	0.024038	0.021635	0.021635	0.019231
0.019231	0.021635	0.024038	0.026709	0.024038	0.021635	0.019231
0.019231	0.021635	0.021635	0.024038	0.021635	0.021635	0.019231
0.019231	0.019231	0.021635	0.021635	0.021635	0.019231	0.019231
0.017361	0.019231	0.019231	0.019231	0.019231	0.019231	0.017361

Tabela 6.2: Matriz de pesos-c do estágio 1

Na Figura 6.6 é apresentada uma representação gráfica da matriz de peso-c utilizada no processamento do estágio 1 e apresentado na Tabela 6.2.

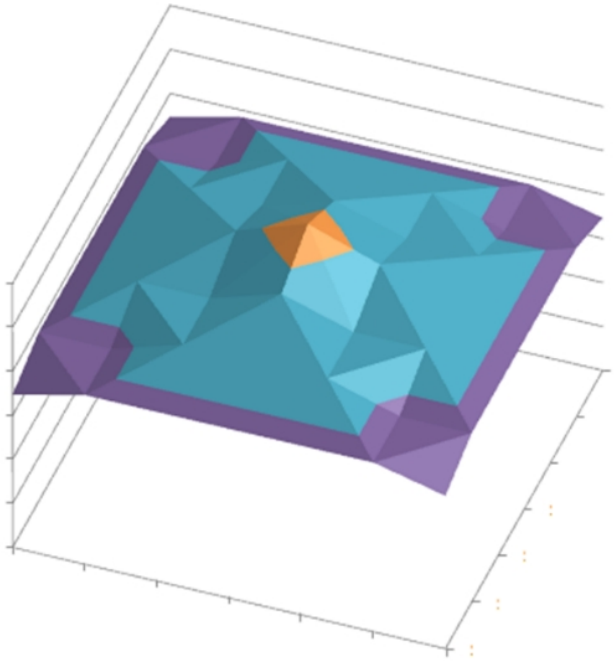

Figura 6.6: Representação gráfica da matriz de peso-c do estágio 1

0.017361	0.019231	0.019231	0.019231	0.019231	0.019231	0.017361
0.019231	0.019231	0.021635	0.021635	0.021635	0.019231	0.019231
0.019231	0.021635	0.021635	0.024038	0.021635	0.021635	0.019231
0.019231	0.021635	0.024038	0.026709	0.024038	0.021635	0.019231
0.019231	0.021635	0.021635	0.024038	0.021635	0.021635	0.019231
0.019231	0.019231	0.021635	0.021635	0.021635	0.019231	0.019231
0.017361	0.019231	0.019231	0.019231	0.019231	0.019231	0.017361

Tabela 6.3: Matriz de Peso-c do Estágio 2

Na Figura 6.7 é apresentada uma representação gráfica da matriz de peso-c utilizada no processamento do estágio 1 e apresentado na Tabela 6.3.

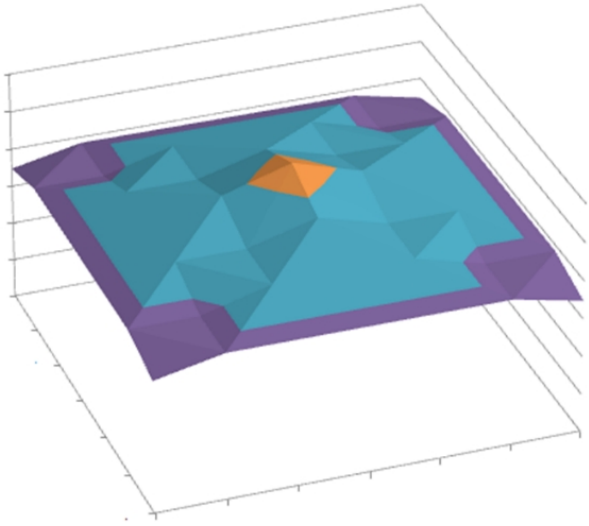

Figura 6.7: Representação gráfica da matriz de peso-c do estágio 2

0.035225	0.039628	0.039628	0.039628	0.035225
0.039628	0.039628	0.044031	0.039628	0.039628
0.039628	0.044031	0.048924	0.044031	0.039628
0.039628	0.039628	0.044031	0.039628	0.039628
0.035225	0.039628	0.039628	0.039628	0.035225

Tabela 6.4: Matriz de peso-c do Estágio-3

Na Figura 6.8 é apresentada uma representação gráfica da matriz de peso-c utilizada no processamento do estágio 1 e apresentado na Tabela 6.4.

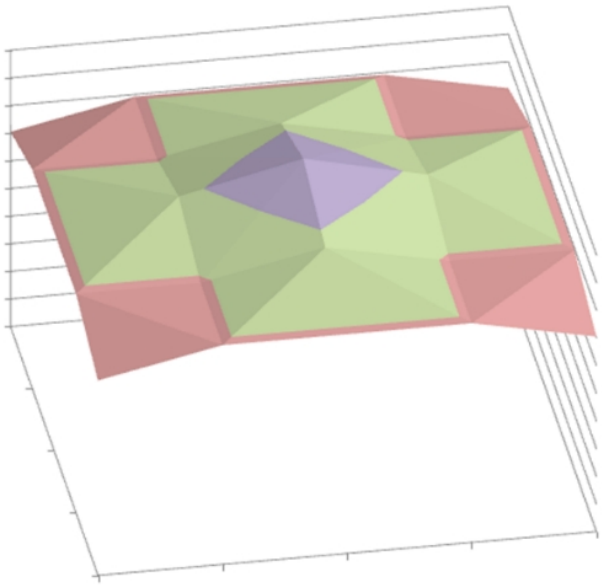

Figura 6.8: Representação gráfica da matriz de peso-c do estágio 3

Nas Tabela 6.5, 6.6 e 6.7 é apresentado as matrizes de peso-d utilizados no processamento dos

estágios 1, 2 e 3 respectivamente.

0.72	0.72	0.72	0.72	0.72
0.72	0.81	0.9	0.81	0.72
0.72	0.9	1	0.9	0.72
0.72	0.81	0.9	0.81	0.72
0.72	0.72	0.72	0.72	0.72

Tabela 6.5: Matriz de peso-d do Estágio 1

Na Figura 6.9 é apresentada uma representação gráfica da matriz de peso-c utilizada no processamento do estágio 1 e apresentado na Tabela 6.5.

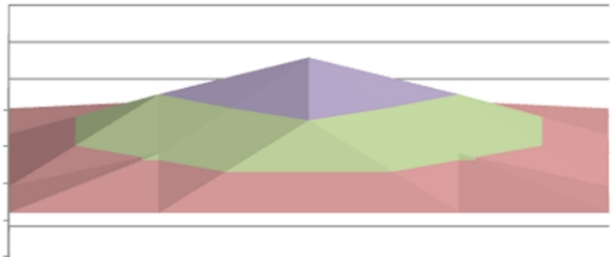

Figura 6.9: Representação gráfica da matriz de peso-d do estágio 1

0.72	0.81	0.81	0.81	0.72
0.81	0.81	0.9	0.81	0.81
0.81	0.9	1	0.9	0.81
0.81	0.81	0.9	0.81	0.81
0.72	0.81	0.81	0.81	0.72

Tabela 6.6: Matriz de peso-d do Estágio 2

Na Figura 6.10 é apresentada uma representação gráfica da matriz de peso-c utilizada no processamento do estágio 1 e apresentado na Tabela 6.6.

0.81	0.9	0.81
0.9	1	0.9
0.81	0.9	0.81

Tabela 6.7: Matriz de peso-d do Estágio 3

Na Figura 6.11 é apresentada uma representação gráfica da matriz de peso-c utilizada no processamento do estágio 1 e apresentado na Tabela 6.7.

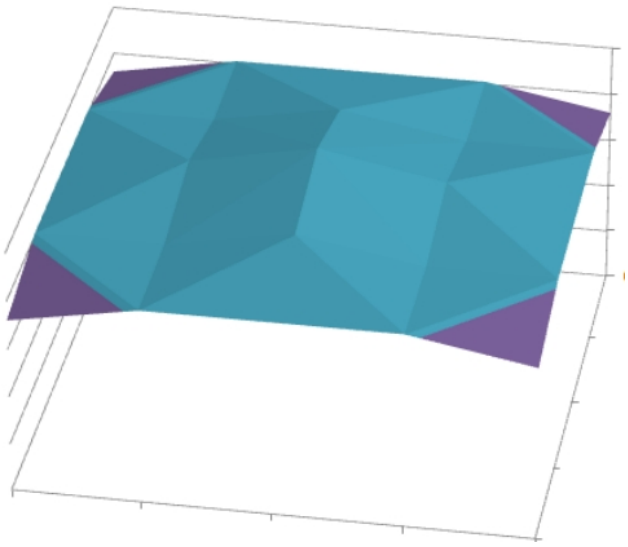

Figura 6.10: Representação gráfica da matriz de peso-d do estágio 2

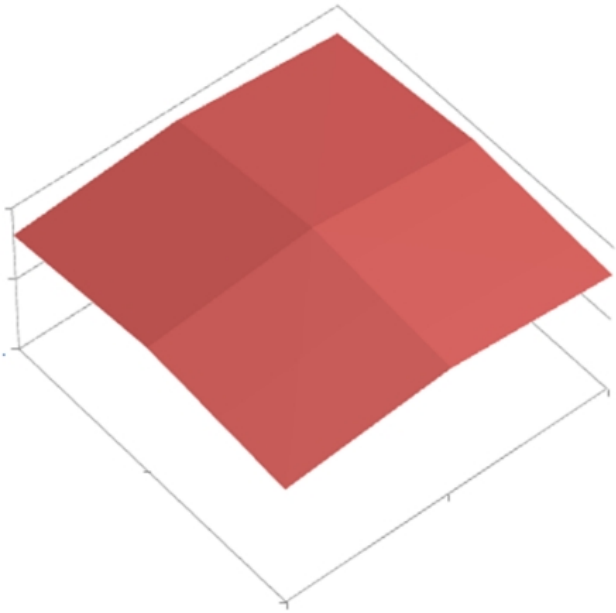

Figura 6.11: Representação gráfica da matriz de peso-d do estágio 3

6.3 Paralelização da Rede Neocognitron dentro do CUDA

Nos Capítulos 3 e 5 foram vistos, respectivamente, a rede Neocognitron e a arquitetura de processamento de dados no GPU com o CUDA. Para se realizar processamento de qualquer tipo de dados dentro do CUDA, é necessário que seja capaz de ser executado dentro de uma organização hierárquica do tipo:

Grade $\longrightarrow$ **Bloco** $\longrightarrow$ ***thread***

A rede Neocognitron possui uma organização igualmente de forma hierárquica, do tipo:

Estágio ⟶ Plano Celular ⟶ Neurônio

Analisando a organização das duas arquiteturas, é possível verificar alguns pontos de similaridade, que podem ser vistos na Figura 6.12 e relacionado na Tabela 6.8, onde vê-se que uma Grade equivale a um Estágio, bem como as conexões de um Plano Celular equivale a um Bloco e o produto do processamento de um bloco de threads equivale a um Neurônio. A equivalência entre as duas arquiteturas facilita a modelagem da rede para ter seu processamento no GPU por meio do CUDA.

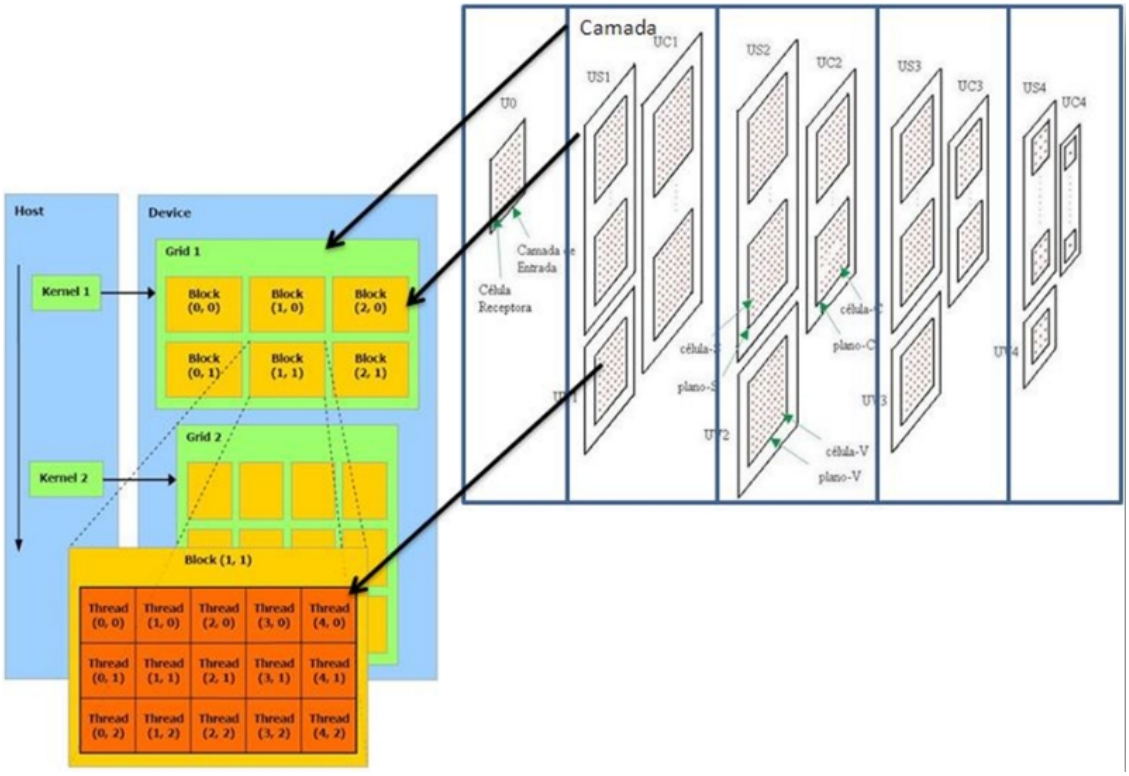

Figura 6.12: Relação entre a arquitetura CUDA e a rede Neocognitron

Um outro fator importante da validade desta equivalência de arquiteturas reside no fato de que existe uma independência dos valores dos neurônios quanto seu processamento. Ou seja, o valor de um neurônio, de um plano celular, não depende do valor do neurônio vizinho no mesmo plano, mas sim, de um processamento maciço dos dados de um plano/estágio anterior. Essa condição valida o uso de uma arquitetura como o GPU/CUDA.

A implementação de um projeto com o uso do GPU/CUDA, determina que existe dois ambientes de processamento; o do host e o dispositivo. Com isso foi desenvolvido um conjunto de funções (Processos) que são executados no *host* e apenas uma função que é executada no dispositivo, como pode ser observada na Figura 6.13.

CUDA	Neocognitron
Grade	Estágio
Bloco	Neurônio
Thread	Conexão

Tabela 6.8: Correspôndencias entre as arquiteturas CUDA x Neocognitron

Este *kernel* tem como responsabilidade o processamento de um único estágio da rede Neocognitron, e é chamado pela aplicação no *host* a cada estágio dentro de uma ordem específica. Deve-se lembrar que a rede modelada neste projeto possui três estágios.

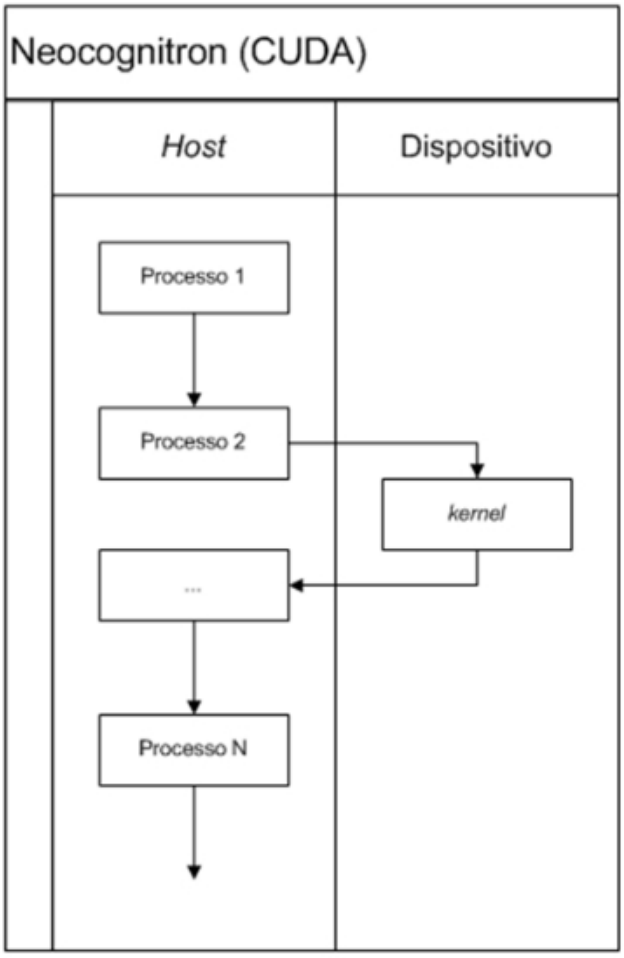

Figura 6.13: Divisão de ambiente quanto o processamento/desenvolvimento de funções

Apesar do processamento da rede ser o mesmo para todos os estágios, a rede Neocognitron apresenta uma redução de dimensionalidade durante sua execução, ou seja, os planos de cada estágio vão reduzindo ao longo do processamento, até o momento em que no Plano-C do último estágio, tenha apenas um único neurônio sendo o número de planos exatamente o número de classes a serem reconhecidas.

Por este motivo o *kernel* é chamado no momento do processamento de um estágio determinado, podendo desta forma, informar ao GPU a configuração específica com relação a dimensão dos blocos de processamento a serem executados.

Os modelos de chamada são apresentados na Tabela 6.9. Para cada estágio levou-se em consideração o tamanho da área de conexão, o objetivo é processar um plano o maior número de áreas de conexão possíveis.

Estágio	Plano-S	ÃĄrea Conexão	Plano-C	Bloco	Gride
1	21x21	7x7	21x21	5x49	1
2	14x14	7x7	14x14	4x49	1
3	7x7	5x5	1x1	10x25	1

Tabela 6.9: Modelo de dimensionalidade por estágio da rede a ser executado

Sendo um bloco processado em um único ciclo do GPU/CUDA, pelos dados da 6.9, tem-se que para o Estágio-1 da rede cada plano possui uma dimensão de 21x21 (441) neurônios a área de conexão deste plano possui uma dimensão de 7x7 (49) conexões. O tamanho do bloco utilizado para o processamento deste estagio foi de 49 *threads*, o mesmo tamanho da área de conexão, sendo que existem 5 blocos dentro da Grade processando simultaneamente, ou seja, são computados 245 conexões simultaneamente, como pode ser observado na Figura 6.14.

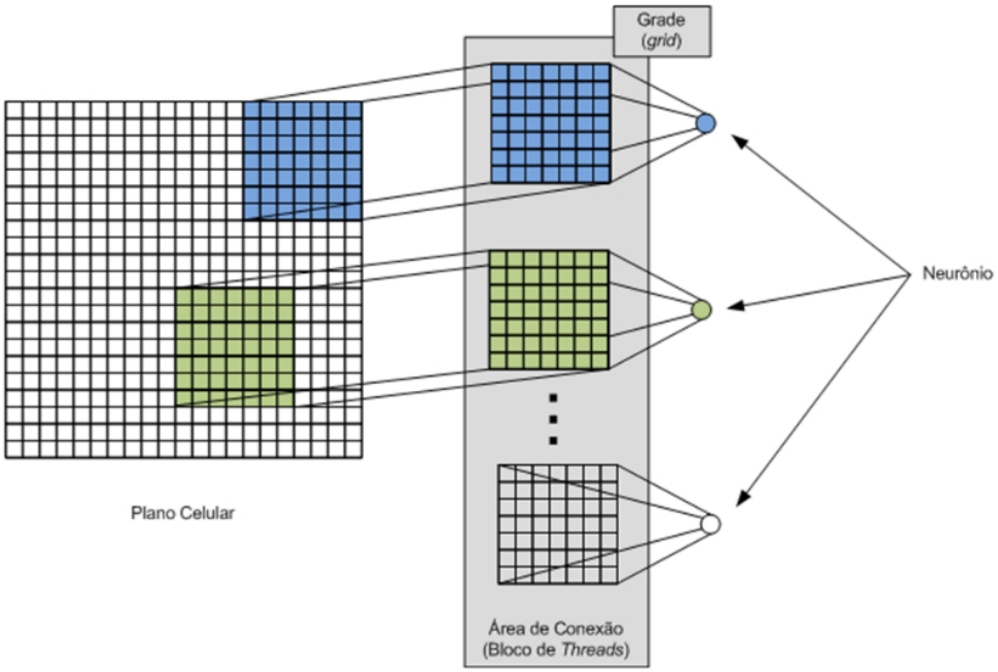

Figura 6.14: Diagrama do processamento das conexões do Neurônio

A mesma análise é válida para os demais estágios da rede apresentados na Tabela 6.9, sendo o número total de neurônios processados simultaneamente de 245, 245 e 250, para os estágios 1, 2 e

3 respectivamente.

6.4 Implementação do Projeto Enquanto Paradigma de HPC

Este projeto usa um GPU (NVIDIA GeForce 8800GTX) como um dispositivo de HPC; o GPU, uma ferramenta de processamento maciço de dados de forma paralela, sendo que este projeto possui um **paralelismo explícito**, uma vez que a responsabilidade pela implementação do código de forma paralela é do desenvolvedor.

Pode-se desta forma classificar o projeto dentro dos paradigmas de processamento de dados em computação de alto desempenho. Esta classificação é descrita nas seções subseqüentes:

6.4.1 HPC/GPU - Segundo seu Fluxo de Instruções de Dados

O GPU apresenta uma arquitetura de processamento do tipo SIMD, uma vez que possui um único fluxo de instruções para um fluxo múltiplo de dados, os quais realizam operações idênticas em dados distintos como pode ser visto no capítulo em "SIMD"e apresentado na Figura 4.3.

6.4.2 HPC/GPU - Tipo de Paralelismo Quanto ao Conteúdo

O GPU apresenta como classificação, segundo o paralelismo de seu conteúdo, a forma "Paralelismo de Dados", uma vez que é executada a mesma instrução sobre dados diferentes, como pode ser visto no capítulo em "Paralelismo de Dados"e apresentado na Figura 4.9.

6.4.3 HPC/GPU - Tipo de Paralelismo Quanto à Forma

O GPU apresenta como classificação, segundo o paralelismo quanto à forma, a "Fases Paralelas", uma vez que os programas desenvolvidos consistem de um número de super-passos e a cada *super-passo* existem duas fases: a computacional e a de interação, como pode ser visto no Capítulo 4, em "Fases Paralelas"e apresentado na Figura 4.12.

6.4.4 HPC/GPU - Granularidade

O nível de paralelismo dentro de um GPU apresenta uma Granularidade Fina, uma vez que a unidade de processamento chega ao nível do neurônio, a menor unidade de processamento possível dentro do projeto (rede Neocognitron).

6.5 Algoritmo e Estrutura de Dados Implementado

Na Figura 6.15 pode-se ver a estrutura de dados utilizada para o armazenamento dos pesos da rede para cada estágio, sendo um vetor onde em cada posição é uma estrutura de dados com dois elementos; uma matriz quadrada 7x7 para armazenar os valores do peso-c e um valor escalar para armazenar o peso-d, ambos do tipo *float*. O número de elementos dentro do vetor é igual ao número de planos do estágio.

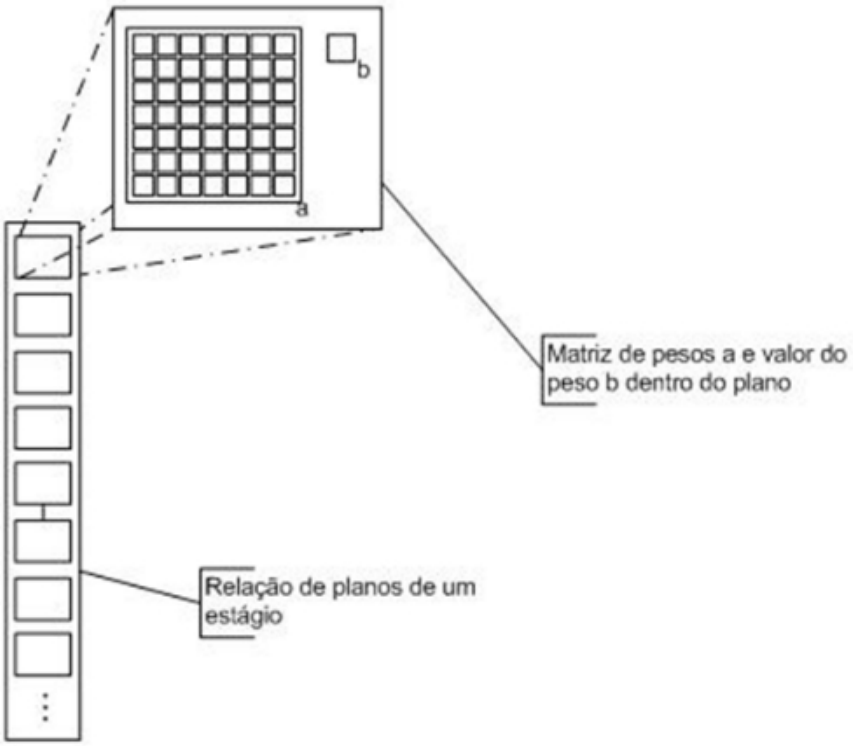

Figura 6.15: Estrutura de dados do estágio A

A informação dos pesos dos estágios gerados pela rede em sua fase de treinamento ficam registrados em um repositório chamado "Rede Treinada"como pode ser observada no diagrama de bloco de funcionamento do sistema da rede apresentado na Figura 6.1.

Para a fase de reconhecimento, objeto deste projeto, foi desenvolvido uma aplicação em C ANSI utilizando a biblioteca de extensão para desenvolvimento de aplicativos de propósito geral em GPU onde cada função a ser executada dentro do dispositivo (GPU) deve ser compilada de forma a gerar um *kernel* (*cubin*) que será executado dentro do dispositivo.

6.6 Organização do Código

Visando um melhor desempenho do processamento da rede, foi implementado um único *kernel* que é chamado a cada processamento de um estágio da rede Neocognitron (três vezes).

A aplicação do host tem a responsabilidade de fazer a carga dos pesos da rede já treinada pela aplicação Delphi; e fazer a transferência dos dados dessa estrutura da memória do host para a memória do dispositivo. Na 6.16 é apresentada a estrutura de dados alocada na memória do dispositivo. Ela utiliza a estrutura de dados apresentada na 6.15, já organizada em três estágios. É adicionada uma variável, do tipo *float*, para armazenar o valor referente a classe vencedora. Este valor é informado no final do processamento da rede (estágio três). Sua existência no projeto é apenas a de facilitar a transferência do valor do resultado entre a memória do dispositivo para a memória do *host*, que sem a sua existência dever-se-ia realizar a transferência de todo o estágio três.

A aplicação (*host*) realiza chamadas ao *kernel*, este é responsável pelo processamento de um estágio da rede. O *kernel* foi desenvolvido de forma a realizar os cálculos do estágio informado no momento de sua chamada. Para a efetivação dos cálculos ele assume como dados de entrada os produzidos pelo processamento do estágio anterior. No caso de ser o primeiro estágio é assumido a camada de entrada.

Sua chamada segue a sintaxe:

CalculaEstagio«<Dg,Db»>(int estagioRef, int estagioUltimo, float valThreshold);

onde *estagioRef* é uma variável do tipo inteiro que indica o número do estágio a ser processado, *estágioUltimo* é uma variável do tipo inteiro e indica o ultimo estágio da rede e *valThreshold* uma variável do tipo *float* que informa o valor do limiar de descrição (*threashold*) que será usado dentro de estágio. Para efeito de processamento, assume-se que o primeiro estágio tem o valor "1"(um), e os planos de cada camada, seguem as regras normais de vetores da linguagem C.

Quando o número do estágio a ser processado (*estagioRef*) for igual ao número de estágios da rede (*estagioUltimo*), o kernel assume que este é o processamento do último estágio e armazena o valor do neurônio referente a classe vencedora a uma variável fora da estrutura da rede. Esta variável é a responsável por apresentar o produto do processamento de toda a rede. Este procedimento pode ser observado dentro da Figura 6.17.

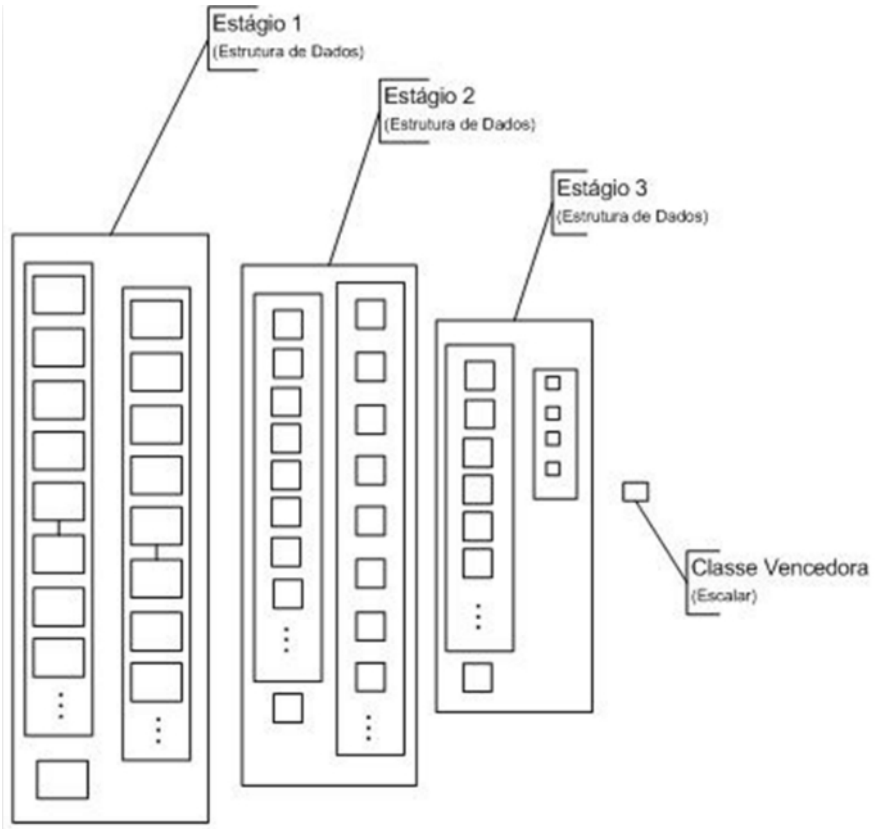

Figura 6.16: Estrutura de Dados alocada dentro do dispositivo

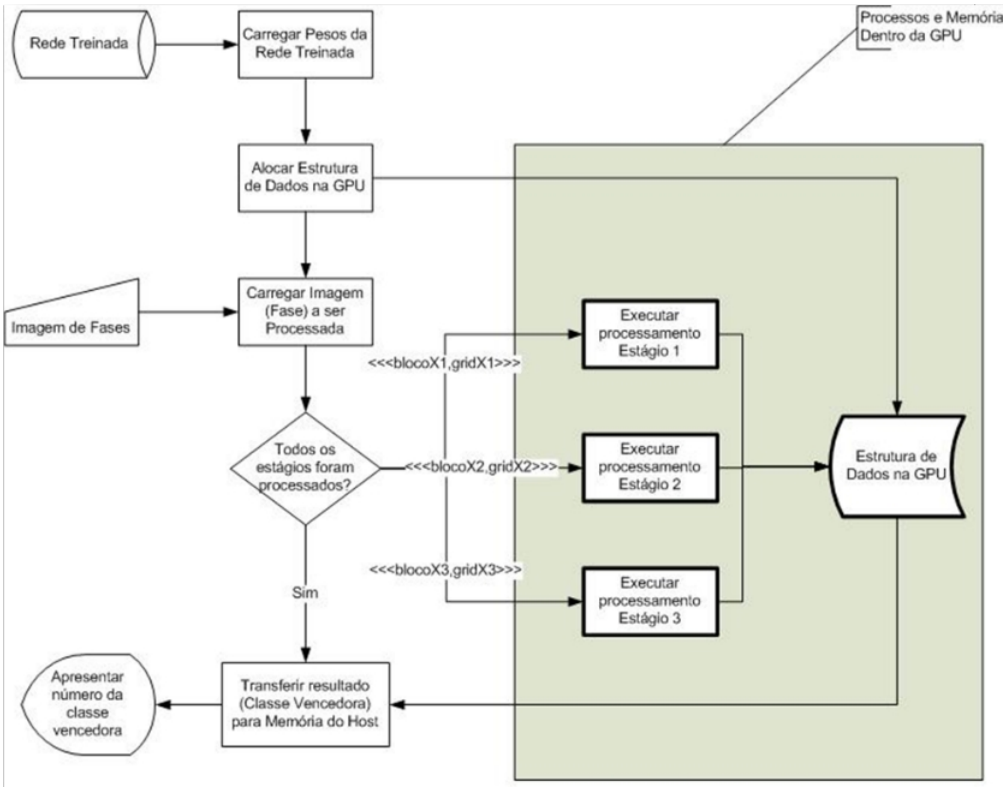

Figura 6.17: Diagrama de blocos da aplicação

6.7 Estágios do Fluxo do Processamento da Aplicação

Nesta seção é descrito o conjunto de elementos que fazem parte do fluxo de processamento da aplicação desenvolvida. Antes da descrição de cada elemento existe uma tabela introdutória onde é apresentado: "Nome do Módulo"que é o nome dado ao módulo dentro do digrama apresentado na Figura 6.17, "Local do Processamento"no dispositivo ou no host e "Tipo do Processamento"se é uma função, um repositório de dados, uma tomada de decisão ou uma saída informativa.

6.7.1 Rede Treinada

- Nome do Módulo: Rede Treinada
- Local: *Host*
- Processamento (Tipo): Repositório

A "Rede Treinada"é um repositório de dados, formado por um conjunto de arquivos binário, residente no disco rígido do computador (host), gerado pela execução do aplicativo Delphi (SAITO et al., 2005). Este repositório é o produto do treinamento da rede Neocognitron, contendo: número de planos de cada estágio, pesos e dados dos planos da camada complexa.

6.7.2 Carga de Pesos da Rede Treinada

- Nome do Módulo: Carregar Pesos da Rede Treinada
- Local: *Host*
- Processamento (Tipo): Função

"Carregar Pesos da Rede Treinada"é uma função, executada no host, responsável por: abrir os arquivos do repositório da "Rede Treinada". Esta função é executada em três estágios distintos, uma para os arquivos relacionados à configuração dos planos (quantidade), pesos (peso-a e peso-b) e plano-C.

Em todas as operações de carga, são respeitadas tanto a formatação da estrutura dos arquivos, utilizada pela aplicação Delphi durante o treinamento, bem como os relacionamentos existentes entre os dados/planos No final desta operação tem-se a estrutura de dados da Figura 6.16, residente na memória do host devidamente populada.

6.7.3 Alocar Estrutura de Dados no GPU

- Nome do Módulo: CAlocar Estrutura de Dados no GPU
- Local: *Host*
- Processamento (Tipo): Função

"Alocar Estrutura de Dados no GPU"é uma função executada no host. Sua responsabilidade é a alocação de espaço na memória do dispositivo e a transferência da estrutura de dados a ser processada (Figura 6.16) da memória do host para a memória do dispositivo.

A alocação da memória do dispositivo se dá por meio do comando cudaMalloc, uma função, equivalente a existente na linguagem C, contudo, o número de bytes a serem reservados (alocados) é na memória do dispositivo. Sua sintaxe é:

```
cudaMalloc( (void**) &d_neocognitron, size));
```

onde *d_neocognitron* é um ponteiro do tipo da estrutura da rede e *size* é uma variável do tipo *unsigned int* que contem o tamanho em *bytes* da estrutura a ser alocada na memória do dispositivo. O prefixo *d_* no ponteiro *d_neocognitron* refere-se que este é um ponteiro para uma área de memória do dispositivo (GPU).

O processo de transferência da estrutura de dedos da memória do host para a memória do dispositivo se dá por meio do documento *cudaMemcpy*, que realiza a cópia de uma posição de memória, indicada por uma variável no *host*, para uma posição de memória indicada no dispositivo. Sua sintaxe é:

```
cudaMemcpy( h_neocognitron, d_neocognitron, size, cudaMemcpyHostToDevice)}
```

onde *h_neocognitron* é um ponteiro para a estrutura da rede alocada na memória do *host*, *d_neocognitron* é um ponteiro para a estrutura da rede alocada na memória do dispositivo, *size* é o tamanho em *bytes* da estrutura da rede a ser copiada e *cudaMemcpyHostToDevice* é uma constante do comando que indica o sentido em que esta sendo realizada a cópia, se do *host* para o dispositivo (*cudaMemcpyHostToDevice*) ou do dispositivo para o *host* (*cudaMemcpyDeviceToHost*).

6.7.4 Estrura de Dados no GPU

- Nome do Módulo: Estrutura de Dados no GPU
- Local: Dispositivo
- Processamento (Tipo): Repositório

"Estrutura de Dados no GPU"é área alocada na memória do dispositivo onde estão, a estrutura de dados da rede, a variável de resposta e o plano de entrada (estágio-0). Esta área é gerada (reservada) pela execução da função "Alocar Estrutura de Dados na GPU".

6.7.5 Imagem de Faces

- Nome do Módulo: Imagem de Faces
- Local: *host*
- Processamento (Tipo): Repositório

"Imagem de Faces"é um repositório em disco rígido do host organizado em um conjunto de arquivos em forma *pgm* (*netpgm grayscale image format*), onde cada arquivo armazena a imagem de uma face humana com a dimensão de 57x57, sendo que cada imagem contem uma face dentro de uma característica distinta, como expressão e iluminação.

Um conjunto dessas imagens foi utilizado para o treinamento, e um segundo grupo agora está sendo usado para o reconhecimento. Existe no grupo de reconhecimento algumas das imagens usadas no momento do treinamento.

6.7.6 Carregar Imagem a ser Processada

- Nome do Módulo: Carregar Imagem a ser Processada
- Local: *host*
- Processamento (Tipo): Função

"Carregar Imagem a ser Processada"é uma função executada no *host* que tem como responsabilidade: abrir um arquivo de imagem, pertencente ao repositório "Imagem de Fases"e carregar a informação de cada *pixel* para uma matriz de dimensão 57x57 e transferir essa matriz para a memória do dispositivo, para o Estágio-0 da rede Neocognitron.

Este procedimento se dá por meio do uso da função *cutLoadPGMf*, que faz a carga de imagem no formato *pgm* para um ponteiro de *floats* na memória do dispositivo. Sua sintaxe é:

```
cutLoadPGMf(face_path, &h_fase, &width, &height);
```

onde *face_path* é o caminho para a imagem que contem a face a ser carregada. Ela deve conter toda a estrutura de diretórios necessária para chegar ao arquivo (caminho absoluto), *h_face* é um ponteiro para *floats* que armazena, na memória do *host*, o conjunto de *pixels* da imagem da face, *width* e *height* são variáveis do tipo *unsigned int* que contem a largura e altura da face, respectivamente.

O valor de *face_path* é informado no momento da chamada, sendo este informado como um parâmetro de chamada como segue a sintaxe a seguir:

```
neocuda /wks/UFSCar/PPG-CC/Data.Base/Image.Face/Students/i0-4.pgm
```

Uma vez que a imagem da face tenha sido carregada ela deve ser transferida para a memória do dispositivo. Isso se dá por meio do uso do comando *cudaMemcpy*.

6.7.7 Processamento dos Estágios da Rede

- Nome do Módulo: Todos estágios foram processados?
- Local: *host*
- Processamento (Tipo): Avaliação

"Todos os estágios foram processados?"é uma avaliação executada pelo *host*, que possui como responsabilidade a chamada do *kernel* para a execução dos cálculos do estágio específico. Neste projeto a rede possui três estágios, sendo que a chamada ao *kernel* é realizada de forma manual (três chamadas consecutivas) como pode ser observado no código a seguir:

```
CalculaEstagio<<<1,49>>>(1, 3, 0.3);
CalculaEstagio<<<1,49>>>(2, 3, 0.3);
CalculaEstagio<<<1,25>>>(3, 3, 0.3);
```

6.7.8 Executar Processamento Estágio X

- Nome do Módulo: Executar Processamento Estágio X
- Local: Dispositivo
- Processamento (Tipo): Função

"Executar Processamento Estágio X"é o *kernel*, a função que é responsável pelo processamento de um estágio da rede, e que é processada pelo dispositivo, ou seja, é enviada no formato de um *cubin* para o *driver* do CUDA que transfere ao *hardware* do GPU que realiza a execução desta função.

Quando o *kernel* for processar o ultimo estágio da rede, este atualiza uma variável, alocada dentro do dispositivo, e que contem o número da classe vencedora do processo de reconhecimento.

6.7.9 Transferir Resultados para Host

- Nome do Módulo: Transferir resultado para Host X
- Local: *host*
- Processamento (Tipo): Função

"Transferir resultado para *Host*"é uma função, executada pelo *host* que após a realização das três chamadas ao *kernel*., faz a cópia do conteúdo da variável que armazena o resultado do processamento (número da classe vencedora) para uma variável alocada dentro da memória do dispositivo, como pode ser observado na sintaxe a seguir:

```
cudaMemcpy( d_winnerClass, h_winnerClass, size, cudaMemcpyoDeviceToHost);
```

onde *d_winnerClass* é a variável alocada na memória do dispositivo, *h_winnerClass* e a variável alocada na memória do *host*, *size* é o tamanho em *bytes* dessa variável e *cudaMemcpyDeviceToHost* é uma constante que indica o sentido da operação de cópia, no caso do dispositivo para o *host*.

6.7.10 Apresentar classe vencedora

- Nome do Módulo: Apresentar classe vencedora X
- Local: *host*
- Processamento (Tipo): Saída Informativa

"Apresentar classe vencedora"é uma função que apresenta em tela (Saída Informativa) o valor da variável que contem o número da classe reconhecida (*d_winneClass*) pelo processamento, como apresentado na sintaxe a seguir:

```
printf( "Winner Class[%d]\n",d_winnerClass);
```

6.8 Ambiente de Desenvolvimento Utilizado

Para a implementação deste projeto foram utilizadas as ferramentas (*software*) da Tabela 6.10 em suas respectivas versões:

Ferramenta	Versão	Sistema operacional
Driver de Video para o CUDA	162.01 (32 bits)	Linux (Fedora Core 8)
SDK CUDA	1.0 (32 bits)	Linux (Fedora Core 8)
CUDA Toolkit	1.0 (32 bits)	Linux (Fedora Core 8)

Tabela 6.10: Relação das Ferramentas utilizadas no projeto

Tendo como ambiente (*hardware*):

- nVIDIA GeForce 8800 GTX 768 MB DRAM 128 Núcles Processadores 1.33 Ghz
- Athlon Duron Duo 2.2 GHz 4 Gb RAM 1.4 Tera Disco

Na Figura 62 é apresentada a foto da estação de trabalho utilizada neste projeto, sendo destacada a presença do GPU e do CPU por meio de setas.

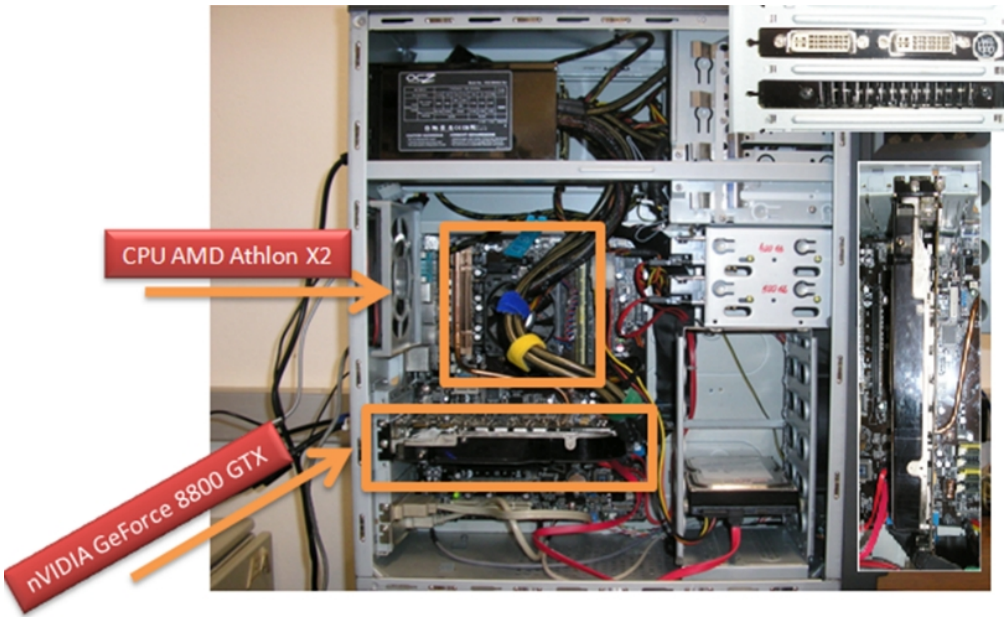

Figura 6.18: Foto da estação de trabalho usada, com destaque a presença do GPU GeForce 8800GTX

6.9 Conclusão

Neste capítulo foi apresentado como a rede Neocognitron foi implementado dentro do GPU com o uso da arquitetura CUDA, bem como o conjunto de ferramentas utilizadas para este procedimento.

No próximo capítulo é apresentado os resultados desta implementação.

7 Resultados

Neste capítulo são apresentados os resultados do processamento da rede Neocognitron dentro da arquitetura do CUDA, cuja implementação foi apresentada no Capítulo 6.

7.1 Acuracidade da Rede Neocognitron Processada no GPU/-CUDA

Nos testes realizados utilizando os repositórios de faces CMU-PIE e o UFSCar (HIRAKURI, 2003), a taxa de reconhecimento obtido durante o reconhecimento é alta e podendo ser aumentado usando mais imagens de treinamento. Os resultados quanto a acuracidade do processamento pode ser visto na Tabela 7.1, onde tempos o nome do repositório de faces e sua taxa, em porcentagem, de acertos.

Repositório de Faces	Taxa de Reconhecimento
CMU-PIE	98%
UFSCar	97%

Tabela 7.1: Taxa de acudarcidade do processamento da rede Neocognitron GPU/CUDA

7.2 Tempos de Execução

Como apresentado no capítulo 6, a rede Neocognitron implementada e executada dentro do GPU, contêm três estágios, na Tabela 7.2 é apresentado o tempo de processamento da rede por estágio.

O tempo total de execução da rede foi de 0,118 segundos.

A medida de tempo foi obtida por meio do uso de funções de controle de tempo de processamento disponibilizadas pelo API do CUDA, sendo utilizadas as funções cutCreateTimer, cutStartTimer, cutStopTimer, cutGetTimerValue e cutDeleteTimer. A seguir são apresentados os códigos

Número do Estágio	Planos da Camada-S	Tempo de Processamento
1	95 planos	0,092 segundos
2	51 planos	0,022 segundos
3	47 planos	0,004 segundos

Tabela 7.2: Tempo de execução por estágio da rede

de obtenção do tempo de processamento, para os estágios da rede:

```
 unsigned int timer = 0;

CUT_SAFE_CALL( cutCreateTimer( &timer));
CUT_SAFE_CALL( cutStartTimer( timer));

CUT_SAFE_CALL(CalculaEstagio<<<1,49>>>(2, 3, 0.3));

CUT_SAFE_CALL( cutStopTimer( timer));

printf("Processing time: %f (ms)\n", cutGetTimerValue( timer));

CUT_SAFE_CALL( cutDeleteTimer( timer));

unsigned int timer = 0;

CUT_SAFE_CALL( cutCreateTimer( &timer));
CUT_SAFE_CALL( cutStartTimer( timer));

CUT_SAFE_CALL(CalculaEstagio<<<1,49>>>(2, 3, 0.3));

CUT_SAFE_CALL( cutStopTimer( timer));

printf("Processing time: %f (ms)\n", cutGetTimerValue( timer));

CUT_SAFE_CALL( cutDeleteTimer( timer));

unsigned int timer = 0;
```

```
CUT_SAFE_CALL( cutCreateTimer( &timer));
CUT_SAFE_CALL( cutStartTimer( timer));

CUT_SAFE_CALL(CalculaEstagio<<<1,25>>>(3, 3, 0.3));

CUT_SAFE_CALL( cutStopTimer( timer));

printf("Processing time: %f (ms)\n", cutGetTimerValue( timer));

CUT_SAFE_CALL( cutDeleteTimer( timer));
```

Por execução da rede, entende-se o conjunto de cálculos necessários para que uma rede Neocognitron, com seus pesos já carregados, possa fazer o reconhecimento de uma imagem que lhe é apresentada. Não estão sendo considerados aqui os tempos de processamento para:

- Fazer a carga dos pesos das camadas, armazenados em arquivos binários;
- Fazer a transferência de dados entre as memórias do dispositivo e do *host*;
- Fazer a carga da imagem da face a ser reconhecida;
- Ou quaisquer outras atividades que estejam fora do escopo de processamento de dados (cálculos e verificações) da rede.

7.3 Desempenho do Ambiente GPU/CUDA

Na Tabela 7.3 é apresentada uma comparação entre o tempo de processamento no GPU/-CUDA, com a mesma rede sendo processada em um ambiente mono processado e em um cluster com 8 processadores valores obtidos em (RIBEIRO, 2002). Foi realizado um ajuste nos tempos objetivos pelo trabalho de (RIBEIRO, 2002) em função da velocidade dos processadores utilizados em seu trabalho e os existentes atualmente. A tabela está organizada em três colunas, onde a primeira possui a "Arquitetura"do ambiente onde foi processado, a segunda "Número de Processadores"utilizados e a terceira a "Velocidade"do processamento que é dado em segundos.

Por meio desta tabela é possível realizar o cálculo do speed-up e da eficiência do processamento entre as arquiteturas. Estes valores são apresentados na Tabela 7.4, organizada em três colunas: na primeira coluna "Arquitetura"indica a arquitetura base considerada para a efetivação do cálculo, a segunda coluna apresenta o valor do speed-up e a terceira coluna a eficiência.

Arquitetura	Número de Processadores	Velocidade (Seg.)
Mono Procssado	1	48
MIMD	8	15
SIMD	128	0,118

Tabela 7.3: Comparação de Tempo de Processamento Entre Arquiteturas

Arquitetura	*Speed-up*	Eficiência
MIMD	79,787	0.623
SIMD	255,319	1.995

Tabela 7.4: Speed-up e Eficiência do SIMD x MIMD

A quantidade total de memória utilizada do dispositivo foi de 437,524 Mb, o que representa uma alocação de 57% da memória total. Sendo este consumo tendo sua distribuição detalhada conforme Tabela 7.5, organizada em duas colunas, a primeira "Área de Reserva"indica onde esta alocada a quantidade de memória em Mega Bytes apresentados na segunda coluna "Espaço Alocado em Mb".

Área de Reserva	Espaço Alocado em Mb
Estágio 1	335,16
Estágio 2	79,968
Estágio 3	9,4
Demais variáveis	12,996

Tabela 7.5: Demonstrativo de áreas alocadas

7.4 Alocação dos Recursos do GPU para o Processamento da Rede

Nesta sessão são apresentados os recursos alocados GPU para a execução da rede em seus estágios. São apresentados três grupos de gráficos onde são apresentados os valores de imagens relacionadas a:

- Variação do uso da memória dentro do *warp*;
- Variação do tamanho dos blocos de *threads* e
- Variação da quantidade de registradores dentro do *thread*.

Estes gráficos são apresentados em três subseções, a seguir, onde estão diferenciados a alocação dos recursos por forma de chamada dos *kernel's* para cada estágio da rede, sendo uma com 245, 245 e 64 *threads* para os estágios 1, 2 e 3 respectivamente.

7.4.1 Recursos Alocados para Processamento do Estágio 01 e 02

A chamada do *kernel* para o cálculo dos estágios um e dois da rede é realizado por meio da sintaxe a seguir:

```
CalculaEstagio<<<1,49>>>(1, 3, 0.3);
CalculaEstagio<<<1,49>>>(2, 3, 0.3);
```

Esta chamada gera um consumo de recursos do dispositivo segundo a Tabela 7.6, onde na primeira coluna são apresentados os recursos alocados para o processamento (*threads* por bloco, registradores por *thread* e memória compartilhada por bloco em *bytes*) e na segunda coluna a quantidade desse recurso alocado para o processamento.

Descrição	Qtd
Threads por bloco 1	245
Registradores por *thread*	10
Memória compartilhada por bloco (*bytes*)	4096

Tabela 7.6: Tabela de recursos usados na chamada ao kernel para calculo do estagio 1 e 2 da rede

A alocação desses recursos pode ser analisada graficamente por meio dos gráficos referentes à Variação do Tamanho do Bloco, Variação do Uso da Memória Compartilhada e Variação da Quantidade de Registradores, sendo estes apresentados a seguir:

A linha azul da 7.1, indica segundo a configuração de recursos alocados uma possível variação do tamanho do bloco. A seta vermelha aponta onde é realmente definido o tamanho do bloco em termos de número de *threads*.

A linha azul da 7.2, indica segundo a configuração de recursos alocados uma possível variação da quantidade de registradores. A seta vermelha aponta onde é realmente definido o número de *threads* por registrador.

A linha azul da 7.3, indica segundo a configuração de recursos alocados uma possível variância do uso da quantidade de memória compartilhada. A seta vermelha aponta onde é realmente definido o tamanho de memória compartilhada por *threads* em *bytes*.

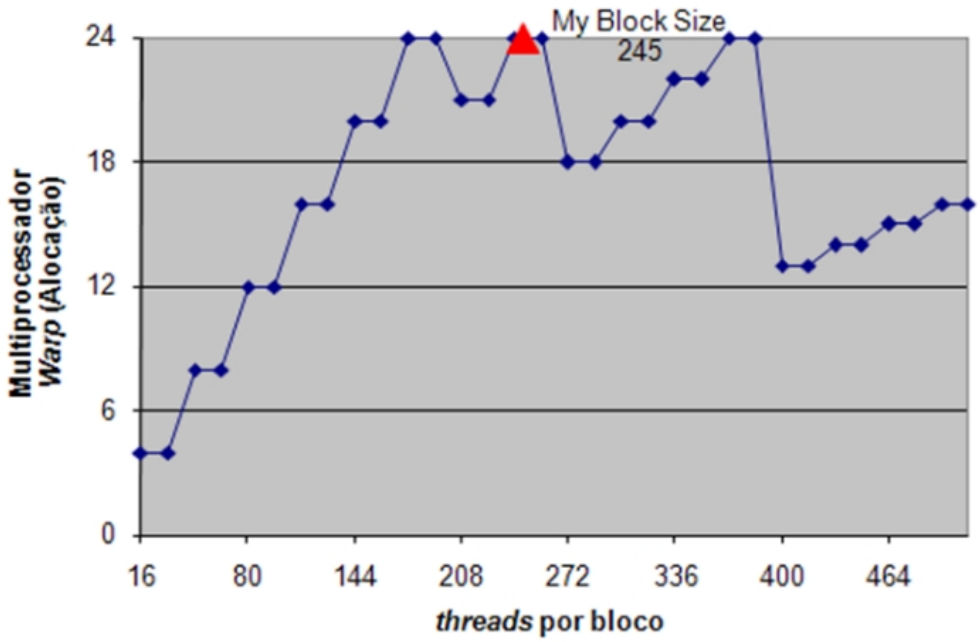

Figura 7.1: Variação do Tamanho do Bloco para chamada ao kernel referente ao cálculo dos estágios 1 e 2

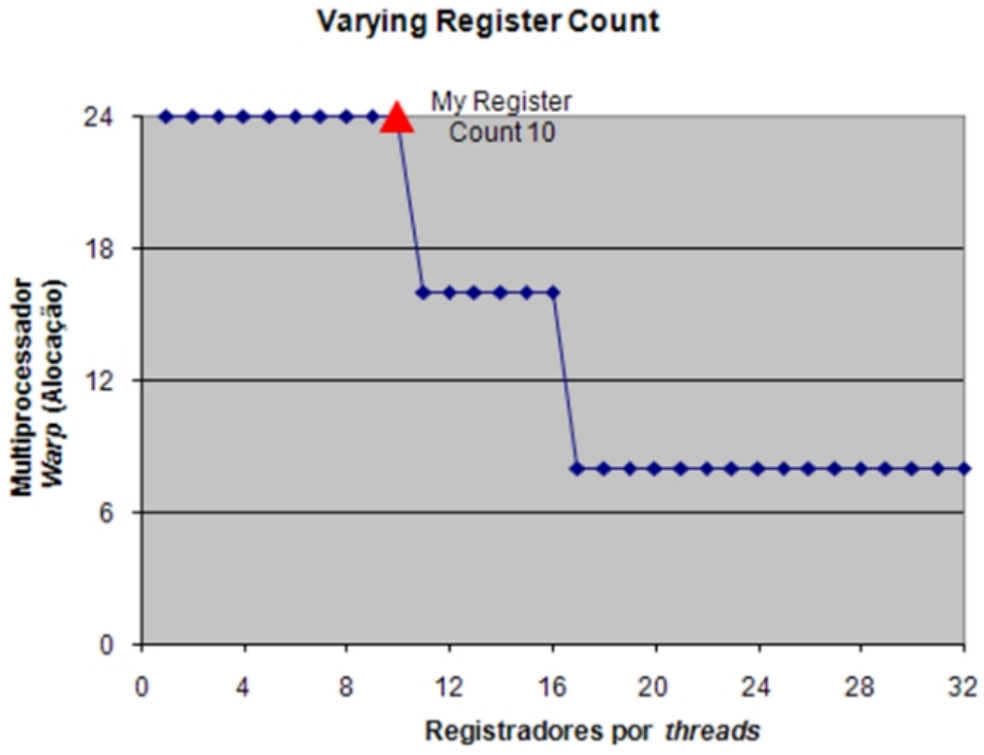

Figura 7.2: Variação da quantidade de registradores para chamada ao kernel referente ao cálculo dos estágios 1 e 2

7.4.2 Recursos Alocados para Processamento do Estágio 03

A chamada do *kernel* para o calculo dos estágios um e dois da rede é realizado por meio da sintaxe a seguir:

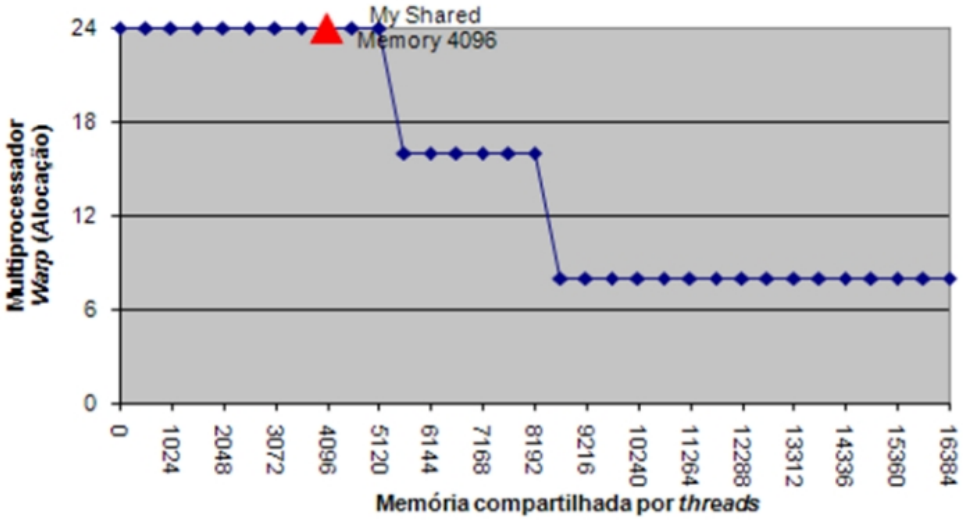

Figura 7.3: Variação da memória compartilhada utilizada para chamada ao kernel referente ao cálculo dos estágios 1 e 2

```
CalculaEstagio<<<1,25>>>(3, 3, 0.3);
```

Esta chamada gera um consumo de recursos do dispositivo segundo a 7.7, onde na primeira coluna são apresentados os recursos alocados para o processamento (*threads* por bloco, registradores por *thread* e memória compartilhada por bloco em *bytes*) e na segunda coluna a quantidade desse recurso alocado para o processamento.

Descrição	Qtd
Threads por bloco	245
Registradores por *thread*	10
Memória compartilhada por bloco (*bytes*)	4096

Tabela 7.7: Tabela de recursos usados na chamada ao kernel para calculo do estagio 3 da rede

A alocação desses recursos pode ser analisada graficamente por meio dos gráficos referentes à Variação do Tamanho do Bloco, Variação do Uso da Memória Compartilhada e Variação da Quantidade de Registradores, sendo estes apresentados a seguir:

A linha azul da 7.4, indica segundo a configuração de recursos alocados uma possível variância do tamanho do bloco. A seta vermelha aponta o tamanho do bloco em termos de números de *threads* usado em 250.

A linha azul da 7.5, indica segundo a configuração de recursos alocados uma possível variância da quantidade de registradores. A seta vermelha aponta o número de registradores usado por *thread*

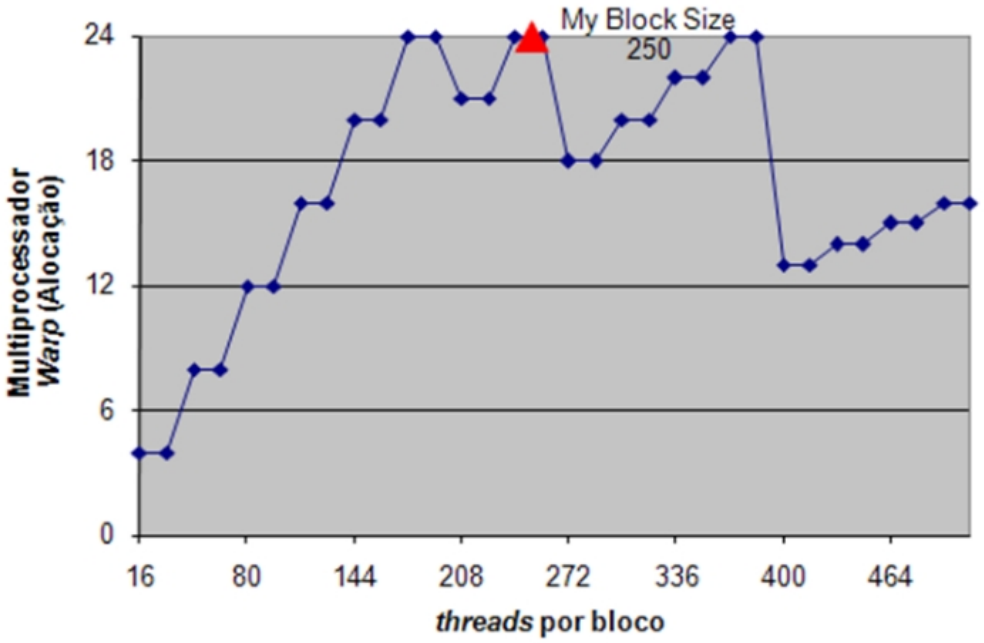

Figura 7.4: Variação do Tamanho do Bloco para chamada ao kernel referente ao cálculo do estágio 3

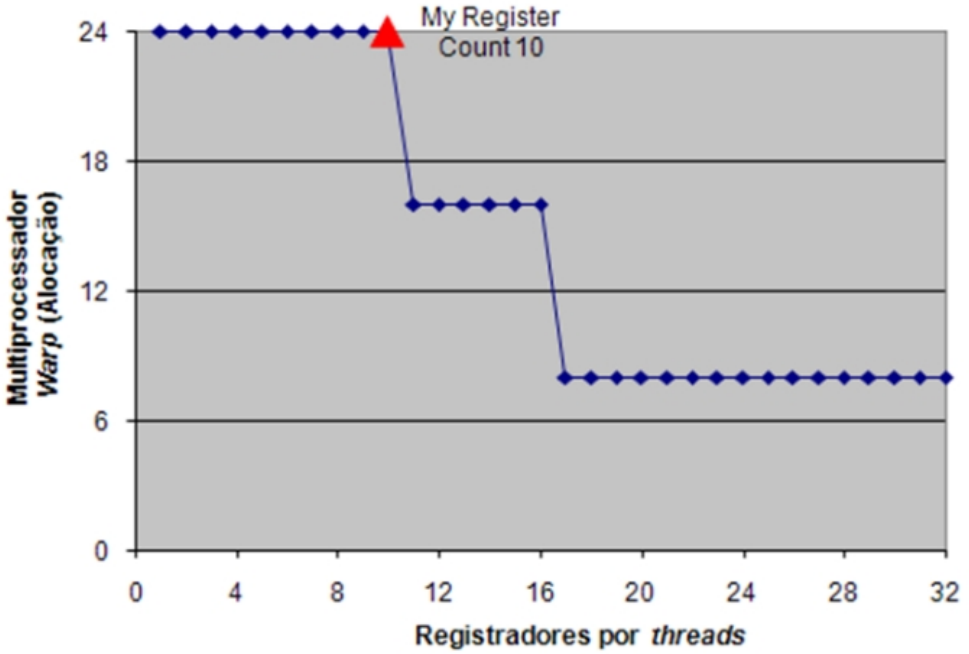

Figura 7.5: Variação da quantidade de registradores para chamada ao kernel referente ao cálculo do estagio 3

como 10.

A linha azul da 7.6, indica segundo a configuração de recursos alocados uma possível variância do uso da quantidade de memória compartilhada. A seta vermelha aponta o número de *bytes* na

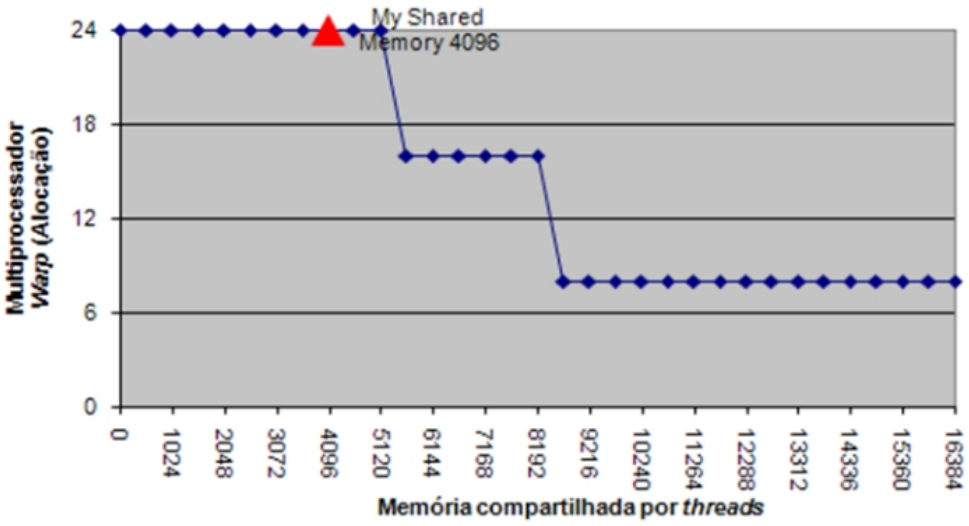

Figura 7.6: Variação da memória compartilhada utilizada para chamada ao kernel referente ao cálculo do estágio 3

memória compartilhada por *threads* como 4098.

7.5 Conclusões

Os resultados apresentados neste capítulo apresentam a arquitetura CUDA como sendo uma arquitetura altamente eficaz para o processamento maciço de dados de forma paralela dentro de um paradigma SIMD.

Todavia os tamanhos das imagns usadas na operação foram pequenas,imagens de dimensão 57 x 57 com o que possibilitou a carga da estrutura total da rede para dentro da memória do dispositivo onde o acesso é protegido e de alta velocidade, fatores que podem ter influenciado nos resultados apresentados.

No próximo capítulo é realizada a apresentação das conclusões que foram obtidas pela análise dos resultados apresentados bem como com a experiência obtida pelo desenvolvimento deste projeto.

8 *Conclusões e Trabalhos Futuros*

8.1 Conclusão do Trabalho

O trabalho de dissertação de mestrado, apresentado, consiste no processamento da rede neural Neocognitron para o reconhecimento facial em ambiente de alto desempenho GPU/CUDA.

Para tanto foi apresentada uma descrição sobre o problema de reconhecimento facial, uma descrição da rede Neocognitron detalhando sua estrutura e funcionalidades. Também foram apresentados os conceitos de computação de alto desempenho e uma revisão dos conceitos de programação paralela, sendo o último ponto da revisão a apresentação do desenvolvimento de aplicações genéricas (científicas) no GPU com o auxilio do CUDA, para que, por fim, fosse apresentada a implementação da rede Neocognitron dentro do GPU.

Com o processamento da rede Neocognitron dentro do GPU/CUDA apresentada neste trabalho, pode-se concluir que houve um significante aumento quanto ao desempenho no processamento para reconhecimento de faces mostrando a viabilidade da utilização deste método.

Numa tentativa de traçar uma linha comparativa entre a rede Neocognitron processada no GPU/CUDA e em arquiteturas tradicionais foi verificada, por meio do cálculo de speed-up uma disparidade, uma vez que se obteve um *speed-up* super linear ($S_p > p$), isso possívelmente ocorreu pelas divergências de arquitetura sendo mais adequado uma comparação entre o processamento GPU/CUDA, com um desenvolvimento "tradicional"no GPU. Mesmo assim foi observado um alto desempenho quando comparado os tempos de processamento.

Uma outra conclusão sobre a implementação de projetos dentro dessas arquitetura, é que esta ameniza alguns problemas comuns existentes quando utilizados outros ambientes de computação paralela, MIMD, por exemplo, onde pode-se citar:

- Sincronização: como a granularidade do desenvolvimento dentro deste dispositivo é baixa e o processamento de dados não concorre por partes da memória (cada *thread* possui seu ponto/área de memória) não existe a necessidade da perda de tempo para a realização de um sincronismo dos processadores;

- Rede: como todo o processamento se dá dentro de um mesmo dispositivo não existe a problemática de quais tipos e recursos de conectividade estarão sendo utilizados, sendo toda a arquitetura do GPU de alta velocidade.

- Contenção: não existe uma concorrência de recursos pelos processadores.

Outro tema onde o GPU apresenta vantagens sobre as arquiteturas tradicionais de computação de alto desempenho (como *cluster*), é com relação ao balanceamento de carga. Como a própria arquitetura de processamento do GPU está focada em processamento de dados do tipo SIMD, projetos que, como a rede Neocognitron tenham seu processamento focados em blocos de dados acabam sendo privilegiados, uma vez que todo um bloco é processado em um único ciclo de instrução.

Contudo, o desenvolvimento de projetos em ambiente GPU/CUDA apresenta como principal dificuldade a modelagem da aplicação enquanto fluxo de processamento dos dados. Conforme visto em outros trabalhos (POLI et al., 2007), não é qualquer aplicação que se beneficia com esta arquitetura. Podendo ser apresentados três categorias de possibilidades de implementação de aplicações no GPU sendo estes: totalmente possíveis de desenvolvimento, parcialmente possíveis de desenvolvimento e inviáveis quanto ao desenvolvimento.

As aplicações que se beneficiam com o processamento em GPU/CUDA, são aquelas que possuem grande volumes de dados, de forma matricial, e que tenham seu processamento independente do valor do resultado de um processamento anterior. O custo computacional para tomadas de decisões é significativo Conclui-se que quanto menor a granularidade dentro de um modelo de dados organizado e estruturado para uma visão de processamento em blocos melhor terá um ganho de desempenho do processamento significativo.

Outro ponto de complexidade, que pode vir a inviabilizar o uso do GPU/CUDA é o não uso de funções proprietárias do API do CUDA.

Embora um sistema biológico seja um sistema naturalmente paralelo e altamente concorrente com um balanceamento de carga impressionante, a consciência humana é basicamente um processo seqüencial, bem como o comportamento dentro da reação aos eventos.

Quando se fala em desenvolvimento de uma solução em ambiente paralelo, o primeiro desafio consiste em pensar uma solução que naturalmente seria implementada de forma seqüencial em uma organização de funções que possam ser executadas em paralelo.

Dentro deste contexto a primeira solução mais acessível é a visão de *clusters* onde os computadores fazem um papel de trabalhadores em um projeto com atividades distintas mas colaborativas

entre si, ou seja, uma arquitetura MIMD. E aqui reside a principal diferença no *software* para um projeto que seja implementado dentro da arquitetura do CUDA.

Software deve estar organizado/focado em BLOCOS de processamento paralelo e não em funcionalidades distribuídas.

8.2 Proposta para Trabalhos Futuros

Com o objetivo de dar continuidade a este trabalho de pesquisa, faz-se a proposta da seguinte relação de trabalhos futuros:

A criação de um ambiente de computação de alto desempenho hibrida

Como foi observado neste trabalho, o uso do GPU como ambiente de computação de alto desempenho, apresenta uma limitação quanto aos tipos de processamento que podem ser executados, tendo seu desempenho diretamente relacionado com a ausência de tomada de decisão durante o fluxo de processamento de dados.

A proposta deste trabalho futuro está em criar uma plataforma híbrida quanto as tecnologias, onde possa ser sanada esta questão, ou seja, partes do projeto, como as quais sejam necessárias tomadas de decisão, durante o fluxo de processamento, poderiam ser executados em outros dispositivos, como o próprio CPU.

A criação de uma biblioteca CUDA com troca de mensagens

Outra proposta de trabalho futuro consiste na criação de uma biblioteca CUDA com a capacidade de troca de mensagens entre outros GPUs. A idéia consiste em usar as funcionalidades da biblioteca MPI, não para troca de mensagens entre processadores, mas sim entre GPUâĂŹs instaladas em diferentes computadores com o objeto de criar um cluster de GPU.

A criação de uma biblioteca de vinculo de dados

ÃL' sabido que o volume de dados em SGBDâĂŹs assume grandes proporções, já não sendo raras as bases de dados com unidade em volta do TeraBytes o objetivo deste trabalho seria o de estender o CUDA, assim como foi realizado com CUFFT e o CUBLAS para uma biblioteca, mais de alto nível, que poderia ser utilizada enquanto ferramenta de apoio a ambientes de mineração de dados.

Processamento de rede Neocognitron para o reconhecimento facial 3D

Os blocos de *threads* que podem ser processadas no GPU/CUDA podem apresentar dimensões de 2D ou 3D. Uma característica natural se for lembrado que esta tecnologia vem evoluindo

segundo as necessidades da área de processamento gráfico onde cenas 3D são uma constante cada vez mais natural.

O objetivo deste trabalho esta em: gerar uma base de faces 3D, estender a rede Neocognitron para o processamento de imagens 3D e sua implementação dentro do GPU/CUDA usando como base de processamento das áreas de conexão cubos 3D.

Referências Bibliográficas

ABU-MOSTAFA, Y.; PSALTIS, D. Optical neural computers. In: *Scientific American*. [S.l.: s.n.], 1987. p. 88–95.

ALMASI, S. G.; GOTTLIEB, A. Highly parallel computing. In: *The Benjamin Cummings Publishing Company Inc.* [S.l.: s.n.], 1994.

BELHUMEUR, P.; HESPANHA, J.; KRIEGMAN, D. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. In: *IEEE Trans. On Pattern Analysis and Machine Intelligence*. [S.l.: s.n.], 1997. p. 711–720.

BIEDERMAN, I.; KALOCSAI, P. Neural and psycophysical analysis of object and face recognition, in face recognition, from theory and applications. In: *Berlin, Springer-Verlag*. [S.l.: s.n.], 1998. p. 3–25.

BRUNELLI, R.; POGGIO, T. Hyperbf networks for gender classification. In: *DARPA Image Understanding Workshop*. [S.l.: s.n.], 1992. p. 311–314.

BUHMANN, J.; LADES, M.; MALSBURG, C. Size and distortion invariant object recognition by hierarchical graph matching. In: *International Joint Conference on Neural Networks*. [S.l.: s.n.], 1990. p. 411–416.

CARRIERO, N.; GELERNTER, D. How to write parallel programs: A guide to the perplexed. In: *ACM Computing Surveys* . [S.l.: s.n.], 1989.

CENTURION, A. M. Análise de desempenho de algoritmos paralelos utilizando plataforma de portabilidade. In: *Dissertação de Mestrado, Instituro de Ciências Matemáticas e de Computação da Universidade de São Paulo (ICMC/USP).* [S.l.: s.n.], 1998.

CHELLAPPA, R.; WILSON, C.; SIROHEY, S. Human and machine recognition of faces: A survey. In: *Proceedings of the IEEE*. [S.l.: s.n.], 1995. p. 705–740.

COMBA, J. L. et al. Computation on gpus: from a programmable pipeline to an efficient stream processor. In: *Revista de InformÃątica TeÃşrica e Aplicada*. [S.l.: s.n.], 2003. p. 41–70.

ETEMAD, L.; CHELLAPPA, R. Discriminant analysis for recognition of human face images. In: *Journal of the Optical Society of America*. [S.l.: s.n.], 1997. p. 1724–1733.

FLYMN, J. M. Some computer organizations and their effectiveness. In: *IEEE TOC*. [S.l.: s.n.], 1972. p. 948–960.

FUKUNAGA, K. Statistical pattern recognition. In: *New York: Academic Press.* [S.l.: s.n.], 1989.

FUKUSHIMA, K. Neural-network model for a mechanism of pattern recognition unaffected by shit in position - neocognitron. In: *Trans. IECE Japan*. [S.l.: s.n.], 1979. p. 656–665.

FUKUSHIMA, K. Neural network model for vision. In: *volume 4*. [S.l.: s.n.], 2003. p. 2625–2630.

FUKUSHIMA, K.; MIYAKE, S. Neocognitron: A new algoriyhm for pattern recognition tolerant of deformations and shift in position. In: *Pattern Recognition*. [S.l.: s.n.], 1982. p. 455–469.

FUKUSHIMA, K.; TANIGAWA, M. Use of different thresholds in learning and recognition. In: *Neurocomputing*. [S.l.: s.n.], 1996. p. 1–17.

FUKUSHIMA, K.; WAKE, N. Improved neocognitronwith bend-detecting cells. In: *IEEE - International Joint Conference on Neural Networks*. [S.l.: s.n.], 1992. p. 190–195.

HIRAKURI, M. H. Aplicação de rede neural neocognitron para reconhecimento de atributos faciais. In: *Dissertação de Mestrado da Universidade Federal de São Carlos (UFSCar)*. [S.l.: s.n.], 2003.

HUBEL, D. H.; WIESEL, T. N. Receptive fields, binocular interaction, and functional architecture in cat's visual cartex. In: *J. of Physiology*. [S.l.: s.n.], 1962. p. 106–154.

HUBEL, D. H.; WIESEL, T. N. Receptive fields and functional architecture of monkey striate cortex. In: *J. of Physiology*. [S.l.: s.n.], 1968. p. 215–243.

HWANG, K.; BRIGGS, A. F. Computer architecture and parallel processing. In: *McGraw-Hill*. [S.l.: s.n.], 1984.

HWANG, K.; XU, Z. Scalable parallel computing. In: *WCB/McGraw-Hill.* [S.l.: s.n.], 1998.

IDG. *A Japonesa Omron desenvolve uma tecnologia de reconhecimento facial por câmera digital em PDA's e telefones celulares*. 2006. "`http://idgnow.uol.com.br/computacao_pessoal/2005/03/03/idgnoticia.2006-03-12.75087885/IDGNoticia_view`".

KIERNER, C. Arquiteturas de sistemas avançados de computação. In: *Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho*. [S.l.: s.n.], 1991. p. 307–353.

KOHONEN, T. Self-organization and associative memory. In: *Berlin: Springer-Verlag.* [S.l.: s.n.], 1988.

KRUGER, N.; POTZSCH, M.; MALSBURG, C. Determination of face position and pose with a learned representation based on labelled graphs. In: *Image and Vision Computing*. [S.l.: s.n.], 1997. p. 665–673.

LADESRAJ, M. et al. Distortion invariant object recognition in the dynamic link architecture. In: *IEEE Trans. On Computers*. [S.l.: s.n.], 1993. p. 300–311.

MANJUNATH, B.; CHELLAPPA, R.; MALSBURG, C. V. A feature based approach to face recognition. In: *IEEE Conference on Computer Vision and Pattern Recognition*. [S.l.: s.n.], 1992. p. 373–378.

MAURER, T.; MALSBURG, C. Single-view based recognition of faces rotated in depth. In: *International Workshop on Automatic Face and Gesture Recognition*. [S.l.: s.n.], 1996. p. 176–181.

MCCREARY, C.; GILL, H. Automatic determination of grain size for efficient parallel processing. In: *Comunication of the ACM New York*. [S.l.: s.n.], 1989. p. 1073–1078.

MOGHDDAM, B.; PENTLAND, A. Probabilistic visual learning for object representation. In: *IEEE Trans. On Pattern Analysis and Machine Intelligence*. [S.l.: s.n.], 1997. p. 696–710.

NAVAUX, A. P. Introdução do processamento paralelo. In: *RBC - Revista Brasileira de Computação*. [S.l.: s.n.], 1989. p. 31–43.

NOTíCIAS, T. *China implanta sistemas de reconhecimento facial biométrico*. 2006. "`http://noticias.terra.com.br/ciencia/interna/0,,OI956783-EI238,00.html`".

NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. [S.l.]: NVIDIA, 2007.

OKADA, K. et al. The bochum/usc face recognition system and how it fared in the feret phase iii. In: *Test. Berlin: Springer-Verlag*. [S.l.: s.n.], 1998. p. 186–205.

PENTLAND, A.; MOGHADDAM, B.; STARNER, T. View-based and modular eigenspaces for face recognition. In: *IEEE Conference on Computer Vision and Pattern Recognition* . [S.l.: s.n.], 1994.

POGGIO, T.; GIROSI, F. Networks for aproximation and learning. In: *Proc. IEEE*. [S.l.: s.n.], 1990. p. 1481–1497.

POLI, G. et al. Voice command recognition with dynamic time warping (dtw) using graphics processing units (gpu) with compute unified device architecture (cuda). In: *SBAC-PAD International Symposium on Computer Architecture and High Performance Computing*. [S.l.: s.n.], 2007. p. 19–27.

QUINN, J. M. Designing efficient algorithms for parallel computers. In: *McGraw Hill.* [S.l.: s.n.], 1987.

RIBEIRO, L. J. Paralelização da rede neural neocognitron em cluster smps. In: *Dissertação de Mestrado da Universidade Federal de São Carlos (UFSCar)* . [S.l.: s.n.], 2002.

SAITO, J. H. et al. Using cmu pie human face database to a convolutional neural network - neocognitron. In: *ESANN2005 - European Symposium on Artificial Neural Networks*. [S.l.: s.n.], 2005. p. 491–496.

SAITO, J. H.; FUKUSHIMA, K. Modular structure of neocognitron to pattern recognition. In: *ICONIP'98 Fifth Int. Conf. On Neural Information and Processing*. [S.l.: s.n.], 1998. p. 279–282.

SEWETS, D.; WENG, J. Using discriminant eigenfeatures for image retrieval. In: *IEEE Trans. On Pattern Analysis and Machine Intelligence*. [S.l.: s.n.], 1996. p. 831–836.

SIM, T.; BAKER, S.; BSAT, M. The cmu pose, illumination, and expression database. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*. [S.l.: s.n.], 2003. p. 1615–1618.

SOUZA, M. A. Avaliação de rotinas de comunicação ponto-a-ponto do mpi. In: *Dissertação de Mestrado, Instituto de Ciências Matemáticas de São Carlos (ICMSC), Universidade de São Paulo (USP).* [S.l.: s.n.], 1996.

SUNDERAM, V. S. et al. The pvm concurrent computing system: evolution experiences and threads. In: *Parallel Computing*. [S.l.: s.n.], 1994. p. 531–545.

TURK, M.; PENTLAND, A. Eigenfaces for recognition. In: *Journal of Cognitive Neuroscience*. [S.l.: s.n.], 1991. p. 72–86.

WISKOTT, L.; FELLOUS, J.; MALSBURG, C. Face recognition by elastic bunch graph matching. In: *IEEE Trans. On Pattern Analysis and Machine Intelligence*. [S.l.: s.n.], 1997. p. 775–779.

ZHAO; CHELLAPPA, R.; PHILLIPS, P. Subspace linear discriminant analysis for face recognition. In: *Technical Report CAR-TR-914, Center for Automation Research, University of Maryland* . [S.l.: s.n.], 1999.

ZHAO, W.; CHELLAPA, R.; KRISHNASWAMY, A. Discriminant analysis of principal components for face recognition. In: *International Conference on Automatic Face and Gesture Recognition*. [S.l.: s.n.], 1998. p. 336–341.

ZHAO, W. et al. Face recognition: A literature survey. In: UNIVERSITY OF MARYLAND. *Technical Report CART-TR-948*. [S.l.], 2002.

APÊNDICE A – Criando Makefile para Desenvolvimento de Projtetos CUDA em Linux

Este apêndice é apresentado como fazer para criar um *Makefile* para o implementação de projetos com o uso do CUDA em ambiente Linux, foi gerado dois arquivos para o automatizar o processo de compilação do projeto em ambiente Linux, o *Makefile*, arquilo tradicional necessário a execução do comando *make* e um segundo arquivo chamado *common.mk* onde são referidos as variáveis de configuração do ambiente CUDA necessários ao processo de compilação.

Uma consideração deve ser realizada, esta sendo considerado que o sdk do CUDA esta instalado no diretório.

/wks/sdk/cuda

```
 ##########################################################################
#
# Copyright 2007 Gustavo Poli Lameirao da Silva.  All rights reserved.
#
##########################################################################
#
# Build script for project
#
##########################################################################

# Add source files here
EXECUTABLE := neocuda
# Cuda source files (compiled with cudacc)
CUFILES := neocuda.cu
```

```
# C/C++ source files (compiled with gcc / c++)
CCFILES := \
   load.cpp \


################################################################
# Rules and targets

include common.mk

1.################################################################
#
# Copyright 2007 Gustavo Poli Lameirao da Silva.  All rights reserved.
#
################################################################
#
# Common build script
#
################################################################

.SUFFIXES : .cu .cu_dbg_o .c_dbg_o .cpp_dbg_o .cu_rel_o .c_rel_o .cpp_rel_o .cubin

CUDA_INSTALL_PATH := /usr/local/cuda
# Basic directory setup for SDK
SRCDIR     ?=
ROOTDIR    ?= /wks/sdk/cuda
ROOTBINDIR ?= $(ROOTDIR)/bin
BINDIR     ?= ../../bin
ROOTOBJDIR ?= obj
LIBDIR     := $(ROOTDIR)/lib
COMMONDIR  := /wks/sdk/cuda/common

# Compilers
NVCC       := nvcc
CXX        := g++
CC         := gcc
```

```
LINK       := g++ -fPIC

# Includes
INCLUDES  += -I. -I$(CUDA_INSTALL_PATH)/include -I$(COMMONDIR)/inc

# OpenGL is used or not (if it is used, then it is necessary to include GLEW)
OPENGLLIB := -lGL -lGLU
ifeq ($(USEGLLIB),1)

# detect if 32 bit or 64 bit system
HP_64 = $(shell uname -i | grep 64)

ifeq "$(strip $(HP_64))" ""
OPENGLLIB += -lGLEW
else
OPENGLLIB += -lGLEW_x86_64
endif
endif

# Libs
LIB       := -L$(CUDA_INSTALL_PATH)/lib -L$(LIBDIR) -L$(COMMONDIR)/lib -lcuda -lcudar

# Warning flags
CXXWARN_FLAGS := \
-W -Wall \
-Wimplicit \
-Wswitch \
-Wformat \
-Wchar-subscripts \
-Wparentheses \
-Wmultichar \
-Wtrigraphs \
-Wpointer-arith \
-Wcast-align \
-Wreturn-type \
-Wno-unused-function \
```

```
$(SPACE)

CWARN_FLAGS := $(CXXWARN_FLAGS) \
-Wstrict-prototypes \
-Wmissing-prototypes \
-Wmissing-declarations \
-Wnested-externs \
-Wmain \

# Compiler-specific flags
NVCCFLAGS :=
CXXFLAGS  := $(CXXWARN_FLAGS)
CFLAGS    := $(CWARN_FLAGS)

# Common flags
COMMONFLAGS = $(INCLUDES) -DUNIX

# Debug/release configuration
ifeq ($(dbg),1)
COMMONFLAGS += -g
NVCCFLAGS   += -D_DEBUG
BINSUBDIR   := debug
LIBSUFFIX   := D
else
COMMONFLAGS += -O3
BINSUBDIR   := release
LIBSUFFIX   :=
NVCC_FLAGS  += --compiler-options -fno-strict-aliasing
CXXFLAGS   += -fno-strict-aliasing
CFLAGS   += -fno-strict-aliasing
endif

# Lib/exe configuration
ifneq ($(STATIC_LIB),)
TARGETDIR := $(LIBDIR)
TARGET   := $(subst .a,$(LIBSUFFIX).a,$(LIBDIR)/$(STATIC_LIB))
```

```
LINKLINE  = ar qv $(TARGET) $(OBJS)
else
LIB += -lcutil$(LIBSUFFIX)
# Device emulation configuration
ifeq ($(emu), 1)
NVCCFLAGS   += -deviceemu
CUDACCFLAGS +=
BINSUBDIR   := emu$(BINSUBDIR)
# consistency, makes developing easier
CXXFLAGS += -D__DEVICE_EMULATION__
CFLAGS += -D__DEVICE_EMULATION__
endif
TARGETDIR := $(BINDIR)/$(BINSUBDIR)
TARGET    := $(TARGETDIR)/$(EXECUTABLE)
LINKLINE  = $(LINK) -o $(TARGET) $(OBJS) $(LIB)
endif

# check if verbose
ifeq ($(verbose), 1)
VERBOSE :=
else
VERBOSE := @
endif

################################################################
# Check for input flags and set compiler flags appropriately
################################################################
ifeq ($(fastmath), 1)
NVCCFLAGS += -use_fast_math
endif

# Add cudacc flags
NVCCFLAGS += $(CUDACCFLAGS)

# Add common flags
NVCCFLAGS += $(COMMONFLAGS)
```

```
CXXFLAGS  += $(COMMONFLAGS)
CFLAGS    += $(COMMONFLAGS)

ifeq ($(nvcc_warn_verbose),1)
NVCCFLAGS += $(addprefix --compiler-options ,$(CXXWARN_FLAGS))
NVCCFLAGS += --compiler-options -fno-strict-aliasing
endif

################################################################
# Set up object files
################################################################
OBJDIR := $(ROOTOBJDIR)/$(BINSUBDIR)
OBJS :=  $(patsubst %.cpp,$(OBJDIR)/%.cpp_o,$(notdir $(CCFILES)))
OBJS +=  $(patsubst %.c,$(OBJDIR)/%.c_o,$(notdir $(CFILES)))
OBJS +=  $(patsubst %.cu,$(OBJDIR)/%.cu_o,$(notdir $(CUFILES)))

################################################################
# Set up cubin files
################################################################
CUBINDIR := $(SRCDIR)data
CUBINS +=  $(patsubst %.cu,$(CUBINDIR)/%.cubin,$(notdir $(CUBINFILES)))

################################################################
# Rules
################################################################
$(OBJDIR)/%.c_o : $(SRCDIR)%.c $(C_DEPS)
$(VERBOSE)$(CC) $(CFLAGS) -o $@ -c $<

$(OBJDIR)/%.cpp_o : $(SRCDIR)%.cpp $(C_DEPS)
$(VERBOSE)$(CXX) $(CXXFLAGS) -o $@ -c $<

$(OBJDIR)/%.cu_o : $(SRCDIR)%.cu $(CU_DEPS)
$(VERBOSE)$(NVCC) -o $@ -c $< $(NVCCFLAGS)

$(CUBINDIR)/%.cubin : $(SRCDIR)%.cu cubindirectory
$(VERBOSE)$(NVCC) -m32 -o $@ -cubin $< $(NVCCFLAGS)
```

```
$(TARGET): makedirectories $(OBJS) $(CUBINS) Makefile
$(VERBOSE)$(LINKLINE)

cubindirectory:
@mkdir -p $(CUBINDIR)

makedirectories:
@mkdir -p $(LIBDIR)
@mkdir -p $(OBJDIR)
@mkdir -p $(TARGETDIR)


tidy :
@find | egrep "#" | xargs rm -f
@find | egrep "\~" | xargs rm -f

clean : tidy
$(VERBOSE)rm -f $(OBJS)
$(VERBOSE)rm -f $(CUBINS)
$(VERBOSE)rm -f $(TARGET)

clobber : clean
rm -rf $(ROOTOBJDIR)
```

APÊNDICE B – 1.1.Exemplo de Código Fonte para Início do Projeto

Nesta seção é apresentado um exemplo de códigos que pode ser utilizado como inicio do desenvolvimento de aplicações dentro da arquitetura CUDA. Uma para a aplicação host e outra para a função *kernel*.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#include <cutil.h>

#include <test_kernel.cu>

void runCUDA( int argc, char** argv);

int
main( int argc, char** argv)
{
    runCUDA( argc, argv);

    CUT_EXIT(argc, argv);
}

void
runCUDA( int argc, char** argv)
{
}
```

```
1.#ifndef _TEST_KERNEL_H_
#define _TEST_KERNEL_H_




#endif
```

Como pode ser observado a função *kernel* nada mais é do um arquivo "h"onde esta sendo implementada uma função específica cujo do arquivo segue uma regra de "_kernel.cu".