UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
Departamento de Engenharia Elétrica

André Márcio de Lima Curvello

Proposta para aceleração de desempenho de algoritmos de visão computacional em sistemas embarcados

São Carlos
2016

André Márcio de Lima Curvello

Proposta para aceleração de desempenho de algoritmos de Visão Computacional em sistemas embarcados

Dissertação apresentada à Escola de Engenharia de São Carlos, Universidade de São Paulo, como parte dos requisitos para obtenção do título de Mestre em Ciências, Programa de Engenharia Elétrica.

Área de Concentração: Processamento de Sinais e Instrumentação

Orientador: Prof. Dr. Evandro Luís Linhari Rodrigues

São Carlos

2016

Trata-se da versão corrigida da dissertação. A versão original se encontra disponível na EESC/USP que aloja o Programa de Pós-Graduação de Engenharia Elétrica.

Curvello, André Márcio de Lima

C975p Proposta para aceleração de desempenho de algoritmos de visão computacional em sistemas embarcados / André Márcio de Lima Curvello; orientador Prof. Dr. Evandro Luis Linhari Rodrigues. São Carlos, 2016.

Dissertação (Mestrado) - Programa de Pós-Graduação em Engenharia Elétrica e Área de Concentração em

Processamento de Sinais e Instrumentação -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2016.

1. Sistemas Embarcados. 2. Linux Embarcado. 3. Visão Computacional. 4. Multicore. 5. GPGPU. 6.

Otimização. I. Título.

FOLHA DE JULGAMENTO

Candidato: Engenheiro **ANDRÉ MARCIO DE LIMA CURVELLO.**

Título da dissertação: "Proposta para aceleração de desempenho de algoritmos de visão computacional em sistemas embarcados".

Data da defesa: 10/06/2015

Comissão Julgadora:	**Resultado:**
Prof. Associado **Evandro Luis Linhari Rodrigues (Orientador)** (Escola de Engenharia de São Carlos/EESC)	APROVADO
Prof. Dr. **Marcelo Andrade da Costa Vieira** (Escola de Engenharia de São Carlos/EESC)	APROVADO
Prof. Dr. **Fernando Santos Osório** (Instituto de Ciências Matemáticas e de Computação/ICMC)	APROVADO

Coordenador do Programa de Pós-Graduação em Engenharia Elétrica:
Prof. Associado **Luis Fernando Costa Alberto**

Presidente da Comissão de Pós-Graduação:
Prof. Associado **Paulo César Lima Segantine**

Dedicatória

Dedico esse trabalho aos meus pais, Dálton Mário e Maria Dalva, que sempre acreditaram em meus sonhos, e se esforçaram ao máximo para me dar todo apoio necessário e fornecer os meios e ferramentas para alcançá-los.

Dedico também à minha querida esposa, Viviane Curvello, minha grande companheira dessa jornada que é a vida, e que esteve ao meu lado em todos os momentos no decorrer deste trabalho, sendo compreensiva com a dedicação necessária e dando-me o apoio, ânimo e incentivo para continuar nos momentos mais difíceis, assim como também se alegrou comigo nos momentos mais celebres.

Dedico também a Deus, autor da Vida.

Agradecimentos

Primeiramente, agradeço a Deus, autor da vida e da minha existência, o início e o fim de todas as coisas.

Agradeço à minha amada esposa, Viviane, por ser uma fortaleza de apoio e consolo, à qual pude recorrer em diversas ocasiões, e que me forneceu o descanso e energia necessária para a realização e conclusão deste trabalho.

Agradeço aos meus pais, que sempre estiveram dispostos e atentos a ajudar e apoiar em tudo quanto precisei. Não teria chegado até aqui sem toda a estrutura que me forneceram.

Agradeço ao Prof. Dr. Evandro Luís Linhari Rodrigues, pela ilustre tarefa de orientação, por todo o apoio que me foi dado, por todas as ideias, críticas e sugestões, que muito ajudaram este trabalho.

Agradeço aos meus amigos do Portal Embarcados, a começar pelo Thiago Lima, por todo o amparo motivacional para o prosseguimento do mestrado, e aos senhores Henrique Rossi, Diego Sueiro e Cleiton Bueno por toda ajuda prestada na parte de Linux Embarcado.

Resumo

CURVELLO, André Márcio de Lima. **Proposta para aceleração de desempenho de algoritmos de visão computacional em sistemas embarcados.** 2016. Dissertação (Mestrado) – Departamento de Engenharia Elétrica e de Computação, Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2016.

O presente trabalho apresenta um *benchmark* para avaliar o desempenho de uma plataforma embarcada WandBoard Quad no processamento de imagens, considerando o uso da sua GPU Vivante GC2000 na execução de rotinas usando OpenGL ES 2.0. Para esse fim, foi tomado por base a execução de filtros de imagem em CPU e GPU. Os filtros são as aplicações mais comumente utilizadas em processamento de imagens, que por sua vez operam por meio de convoluções, técnica esta que faz uso de sucessivas multiplicações matriciais, o que justifica um alto custo computacional dos algoritmos de filtros de imagem em processamento de imagens. Dessa forma, o emprego da GPU em sistemas embarcados é uma interessante alternativa que torna viável a realização de processamento de imagem nestes sistemas, pois além de fazer uso de um recurso presente em uma grande gama de dispositivos presentes no mercado, é capaz de acelerar a execução de algoritmos de processamento de imagem, que por sua vez são a base para aplicações de visão computacional tais como reconhecimento facial, reconhecimento de gestos, dentre outras. Tais aplicações tornam-se cada vez mais requisitadas em um cenário de uso e consumo em aplicações modernas de sistemas embarcados. Para embasar esse objetivo foram realizados estudos comparativos de desempenho entre sistemas e entre bibliotecas capazes de auxiliar no aproveitamento de recursos de processadores *multicore*. Para comprovar o potencial do assunto abordado e fundamentar a proposta do presente trabalho, foi realizado um *benchmark* na forma de uma sequência de testes, tendo como alvo uma aplicação modelo que executa o algoritmo do Filtro de Sobel sobre um fluxo de imagens capturadas de uma webcam. A aplicação foi executada diretamente na CPU e também na GPU embarcada. Como resultado, a execução em GPU por meio de OpenGL ES 2.0 alcançou desempenho quase 10 vezes maior com relação à execução em CPU, e considerando tempos de *readback*, obteve ganho de desempenho total de até 4 vezes.

Palavras-chave: Sistemas embarcados. Linux embarcado. Visão Computacional. *Multicore*. GPU. OpenGL ES 2.0.

Abstract

CURVELLO, André Márcio de Lima. **Proposta para aceleração de desempenho de algoritmos de visão computacional em sistemas embarcados.** 2016. Dissertação (Mestrado) – Departamento de Engenharia Elétrica e de Computação, Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2016.

This work presents a benchmark for evaluating the performance of an embedded WandBoard Quad platform in image processing, considering the use of its GPU Vivante GC2000 in executing routines using OpenGL ES 2.0. To this goal, it has relied upon the execution of image filters in CPU and GPU. The filters are the most commonly applications used in image processing, which in turn operate through convolutions, a technique which makes use of successive matrix multiplications, which justifies a high computational cost of image filters algorithms for image processing. Thus, the use of the GPU for embedded systems is an interesting alternative that makes it feasible to image processing performing in these systems, as well as make use of a present feature in a wide range of devices on the market, it is able to accelerate image processing algorithms, which in turn are the basis for computer vision applications such as facial recognition, gesture recognition, among others. Such applications become increasingly required in a consumption and usage scenario in modern applications of embedded systems. To support this goal were carried out a comparative studies of performance between systems and between libraries capable of assisting in the use of multicore processors resources. To prove the potential of the subject matter and explain the purpose of this study, it was performed a benchmark in the form of a sequence of tests, targeting a model application that runs Sobel filter algorithm on a stream of images captured from a webcam. The application was performed directly on the embbedded CPU and GPU. As a result, running on GPU via OpenGL ES 2.0 performance achieved nearly 10 times higher with respect to the running CPU, and considering readback times, achieved total performance gain of up to 4 times.

Keywords: Embedded Systems. Embedded Linux. Computer Vision. *Multicore*. GPU. OpenGL ES 2.0.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas, Siglas, Unidades e Símbolos

API	-	Interface de Programação de Aplicação – *Application Programming Interface*
ABI	-	Application Binary Interface
CPU	-	Unidade de Processamento Central - *Central Processing Unit*
DMA	-	Direct Memory Access
EABI	-	Embedded Application Binary Interface
CPU	-	Central Processing Unit
GPU	-	Graphics Processing Unit
GPGPU	-	General Purpose Graphics Processing Unit
GB	-	Giga Byte
GHz	-	Giga Hertz
HDMI	-	Interface Multimídia de Alta-Definição - *High-Definition Multimedia Interface*
LAVISIM	-	Laboratório de Visão Computacional e Sistemas Microprocessados
MIPS	-	Microprocessador sem estágios de *pipeline* intertravados - *Microprocessor without Interlocked Pipeline*
MHz	-	MegaHertz
MB	-	MegaBytes
MP	-	Mega Pixels
OpenCL	-	Open Computing Language
OpenCV	-	Open Computer Vision
OpenGL	-	Open Graphics Library
OpenGL ES	-	Open Graphics Library for Embedded Systems

OpenMP	-	Open Multi-Processing
OpenVG	-	Open Vector Graphics
SoC	-	Sistema em um *Chip* - *System on a Chip*
RAM	-	Memória de Acesso Aleatório - *Random Access Memory*
RISC	-	Computação com conjunto reduzido de instruções - *Reduced Instruction set computing*
TBB	-	Threading Building Blocks
VFP	-	Vector Floating Point

“Acho que todo mundo nesse país deveria aprender a programar um computador, porque isso te ensina como pensar.”

— Steve Jobs

Sumário

1. Introdução

O presente trabalho apresenta um *benchmark* para avaliar o desempenho de uma plataforma embarcada WandBoard Quad no processamento de imagens, considerando o uso da sua GPU[1] Vivante GC2000 na execução de rotinas usando OpenGL ES 2.0[2]. Para esse fim, foi tomado por base a execução de filtros de imagem em CPU[3] e GPU. Os filtros são as aplicações mais comumente utilizadas em processamento de imagens, que por sua vez operam por meio de convoluções, técnica esta que faz uso de sucessivas multiplicações matriciais, o que justifica um alto custo computacional dos algoritmos de filtros de imagem em processamento de imagens.

Dessa forma, o emprego da GPU em sistemas embarcados é uma alternativa que torna viável a realização de processamento de imagem nestes sistemas, pois além de fazer uso de um recurso presente em uma grande gama de dispositivos presentes no mercado, é capaz de acelerar a execução de algoritmos de processamento de imagem, que por sua vez são a base para aplicações tais como reconhecimento facial, reconhecimento de gestos, dentre outras. Tais aplicações tornam-se cada vaz mais requisitadas em um cenário de uso e consumo em aplicações modernas de sistemas embarcados.

É crescente e bem perceptível a disseminação de dispositivos móveis tais como celulares *smartphones* e *tablets*, que chegam a possuir um poder computacional equivalente a computadores de dez anos atrás. Para efeito de comparação, um *smartphone* Samsung Galaxy S6[4] vem equipado com um processador Exynos 7420, o qual possui 8 núcleos que operam em velocidades entre 1.5 GHz e 2.1 GHz, somado a 3 GB de memória RAM. Em contrapartida, um computador típico de 2004 poderia ser um Pentium 4 com frequência de operação de 2.8 GHz e 1 GB de memória RAM, com a diferença que o primeiro cabe em um bolso, outro ocupa boa parte de uma mesa.

---

[1] GPU é a abreviação de *Graphics Processing Unit* – Unidade de Processamento de Gráficos. É o processador responsável por realizar cálculos com vetores, matrizes, cenários, etc. Fonte: http://www.webopedia.com/TERM/G/GPU.html

[2] OpenGL ES 2.0 é a versão 2.0 da biblioteca para gráficos OpenGL voltada para sistemas embarcados. Fonte: https://www.khronos.org/opengles/2_X/

[3] CPU é a abreviação de *Central Processing Unit* – Unidade de Processamento Central. É o principal processador de um sistema, executando as instruções de programas. Fonte: http://www.webopedia.com/TERM/C/CPU.html

[4] Informações bem detalhadas do aparelho podem ser vistas no *site* GSM Arena, que também fornece informações detalhadas a respeito de toda gama de dispositivos móveis, sejam *smartphones* ou *tablets* do mercado. Sobre o Samsung Galaxy S6, o *link* para acesso é este: http://www.gsmarena.com/samsung_galaxy_s6-6849.php

De acordo com um relatório liberado pelo Gartner Group[5], só em 2013 foram vendidos cerca de 179 milhões de *tablets* e cerca de 1.890 bilhão de celulares, envolvendo dispositivos com sistemas Android, iOS, Windows Phone, dentre outros[6].

Grande parte destes dispositivos vêm equipados com processadores ARM, e uma parcela cada vez maior faz uso de sistemas operacionais Linux e/ou derivados, como Android e FireFox OS. A título de exemplo, já em 2002, dispositivos móveis com processadores ARM correspondiam a 70 % do total[7] em circulação no mercado.

Apesar destes processadores estarem cada vez mais disponíveis ao público em geral, o uso otimizado de seus recursos adicionais envolve a aplicação de rotinas, diretivas de compilação e de códigos específicos para a arquitetura, além de bibliotecas adicionais. Este uso otimizado incorre na leitura e conhecimento aprofundado da arquitetura alvo. Como se não bastasse, agora já não é mais tão somente necessário conhecer a arquitetura da CPU, mas do SoC como um todo, envolvendo CPU, GPU e Coprocessadores.

Com o crescente aumento do uso de dispositivos móveis, também há o aumento na procura por oferta de novas funcionalidades, que agreguem facilidades ao uso dos dispositivos, e um consequente atrativo ao uso do produto, para favorecer sua venda e adoção no mercado.

Uma destas facilidades é o uso de recursos visuais, e como exemplo já são relativamente comuns a venda e o uso de dispositivos *smartphones* que permitem desbloqueio do sistema por meio de reconhecimento facial, e outros que permitem o controle do aparelho através de gestos. E de acordo com o grupo *Embedded Vision Alliance*, a tendência é a de que estas aplicações tenderão cada vez mais a se disseminar (WILSON e DIPERT, 2013).

As rotinas de reconhecimento facial, reconhecimento de gestos, dentre outras, são executadas por sofisticados algoritmos de Visão Computacional que demandam um elevado poder de processamento, e uma peça-chave fundamental para um produto que faz uso de Visão Computacional ser viável é que funcione com um tempo de resposta tão rápido quanto se queira. Ou seja, a aplicação deve ter um tempo de resposta aos estímulos visuais tão rápida quanto se queira, e isso é um desafio quando se trata de sistemas embarcados.

Um dos principais motivadores deste trabalho foi o artigo de Wilson e Dipert (2013), publicado pela *Embedded Vision Alliance* e intitulado *Embedded Vision on Mobile Devices* –

---

[5] Gartner Group é um ótimo portal para acessar estatísticas e estimativas para o mercado de Tecnologia da Informação. Detalhes sobre o mercado móvel para o ano de 2014 são mostrados neste *link*: http://www.gartner.com/newsroom/id/2645115

[6] Notícia disponível no *site* IDC, e pode ser acessada no seguinte *link*: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

Oportunities and Challenges. Neste artigo, os autores discorrem que, da mesma forma que tecnologias de comunicação *wireless* se agregaram aos equipamentos nos últimos 10 anos, assim também tecnologias de visão computacional serão nos próximos 10 anos. O artigo também cita uma notícia da *ABI Research*[8], que estipula que até 2017 cerca de 600 milhões de celulares *smartphones* terão funcionalidades de controle gestual implementados por meio de visão computacional.

Wilson e Dipert (2013) também destacam que as rotinas que envolvem tarefas de processamento de imagens são computacionalmente pesadas, e que CPUs tradicionais não são otimizadas para processamento de imagens, quanto mais CPUs de dispositivos móveis, o que resulta em uma busca por novos arranjos arquiteturais para aumentar a performance computacional e a eficiência energética dos processadores de sistemas móveis, como a inclusão de coprocessadores auxiliares para tarefas de processamento multimídia, por exemplo. E a despeito de já existirem processadores com CPU, GPU e até mesmo unidade DSP integrados, é de se esperar também a incorporação com núcleos específicos para processamento de imagem.

A fronteira da execução de aplicações com Visão Computacional envolve o uso de rotinas de Processamento de Imagens, que são responsáveis por preparar as imagens a serem trabalhadas para os sofisticados algoritmos de Visão Computacional. Por ser então a fronteira do processo, uma melhoria na sua execução é capaz de acelerar a aplicação como um todo.

Dessa forma, este trabalho apresenta uma proposta para acelerar a execução de rotinas de processamento de imagens em sistemas embarcados com Linux com foco na execução de códigos de Processamento de Imagens na GPU do sistema, com base em características que serão melhor apresentadas no decorrer do trabalho.

1.1. Motivação

Por consumirem muito menos energia do que computadores convencionais, sistemas ARM modernos estão sendo extremamente visados como plataformas para sistemas de processamento de dados e servidores.

---

[8] *600 Million Smartphones Will Have Vision-Based Gesture Recognition Features in 2017*. Artigo publicado pela *ABI Research*, disponível em https://www.abiresearch.com/press/600-million-smartphones-will-have-vision-based-ges.

Pelo seu baixo consumo, várias plataformas dispensam inclusive o uso de *coolers* para a refrigeração do processador. Juntando-se isso ao tamanho reduzido que tais sistemas podem possuir, principalmente quando comparados aos tradicionais sistemas Desktop, é possível agrupar uma grande quantidade de processadores SoC no espaço que uma CPU convencional ocuparia.

Na Figura 1 é possível ver um rack de servidores ARM desenvolvido pela HP em conjunto com a empresa Calxeda[9], uma empresa sediada em Austin, EUA, que desenvolveu um SoC ARM de alto desempenho que consome 5 Watts de energia. O modelo mostrado na Figura 1 pode comportar até 72 nós, ou 72 unidades computacionais independentes.

Figura 1 - Raque de processadores ARM com foco em aplicações servidoras. Fonte: theregister.com

Outro exemplo que poderia ser citado é o do Prof. Simon Cox, da Universidade de Southampton, que montou um *cluster* com 64 Raspberry PI[10], cada uma com um cartão SD de 16GB, gastando o equivalente a quatro mil dólares, sem contar os *switches* necessários para as conexões *Ethernet* entre cada placa. O objetivo do projeto do Prof. Cox é avaliar as funcionalidades do cluster de Raspberry PI, mostrado na Figura 2, no desempenho de atividades computacionais pesadas, fazendo o conjunto funcionar como um pequeno supercomputador.

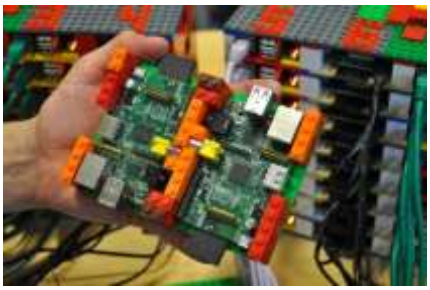


Figura 2 - Imagem de um par de Raspberry PI em um cluster montado no MIT. Fonte: www.arstechnica.com

---

[9] Notícia completa no artigo *HP Project Moonshot hurls ARM servers into the heavens*, disponível em http://www.theregister.co.uk/2011/11/01/hp_redstone_calxeda_servers/, que destaca o surgimento de servidores ARM com foco em baixo consumo de energia.

[10] É possível acessar a própria página do projeto, escrita pelo Prof. Simon Cox, onde o mesmo orienta a reprodução do *cluster* de Raspberry Pi, neste *link*: https://www.southampton.ac.uk/~sjc/raspberrypi/pi_supercomputer_southampton.htm

Recentemente, a Samsung lançou a aplicação *Power Sleep*[11] na loja virtual *Google Play*. A aplicação foi desenvolvida em conjunto com a Universidade de Viena, com o objetivo de usar os recursos computacionais avançados dos *smartphones* atuais para ajudar a quebrar a sequência de proteínas na pesquisa de soluções contra o câncer e Mal de Alzheimer.

O foco da aplicação *Power Sleep*, mostrada na Figura 3, é ser usada quando os aparelhos estão em período de ociosidade, principalmente no período noturno, quando a grande maioria dos usuários deixa seus aparelhos plugados na tomada, para carregar suas baterias. Neste período, e contando com uma conexão com a *Internet*, todo o sistema atua como um *grid* computacional, em que vários nós desempenham tarefas computacionais distribuídas.

Figura 3 - Tela do vídeo demonstrativo da aplicação Samsung Power Sleep. Fonte: Samsung.com.

De sobremodo também recente, a empresa Google anunciou um novo projeto, chamado Tango, que nada mais é do que um *smartphone* com sensores e processadores especiais para mapeamento de ambientes com profundidade em 3D. Para tal, o aparelho conta com um processador adicional especialmente projetado para visão computacional, chamado Myriad 1, que agrega novas instruções para tratar imagens e lidar com dados obtidos dos sensores do *smartphone*. Veja um exemplo do sistema em operação na Figura 4.

Figura 4 - Foto do protótipo do Projeto Tango, desenvolvido pela empresa Google, à esquerda, e um demonstrativo do aparelho em funcionamento, à direita. Fonte: Google.com.

[11] Aplicação Samsung Power Sleep, disponível para aparelhos *smartphones* com sistema operacional Android, pode ser acessado neste *link*: https://play.google.com/store/apps/details?id=at.samsung.powersleep

Outro projeto inovador da mesma empresa é o *Google Glass*, que basicamente incorpora o *hardware* de um *smartphone*, ao possuir um SoC OMAP 4430, da *Texas Instruments*, que agrega um processador *dual-core* ARM Cortex-A9, de 1 GHz, 1 GB de RAM e DSP c64x integrado, além de sensores como acelerômetro, giroscópio, microfone, magnetômetro, sensor de luminosidade e de proximidade, e uma câmera com resolução de 5 MP. A interação com o usuário é feita por meio de comandos de voz, e o usuário interage com o sistema por meio de um *display* projetado no olho direito.

O objetivo do *Google* com o *Google Glass* é desenvolver uma nova gama de aplicações em que o usuário pode ter maior interação com o ambiente à sua volta, como obter auxílios durante o preparo de uma receita culinária, saber o percurso de um trajeto projetado em seu campo de visão, obter informações de locais por meio de Realidade Aumentada, tirar fotos ou filmar por meio de comandos rápidos, dentre outros. Uma foto do *Google Glass* pode ser vista na Figura 5, que também mostra um exemplo do mesmo em funcionamento, da perspectiva do usuário.

Figura 5 - Foto do Google Glass, à esquerda, e imagem ilustrativa de seu uso, da perspectiva do usuário, à direita. Fonte: Google.com.

Os projetos apresentados mostram que sistemas embarcados com processadores ARM consomem pouca energia e podem ser aplicados desde em processamento de propósito geral, até mesmo em aplicações que demandam recursos avançados de visão computacional como Project Tango e o Google Glass.

Componentes agregados em um SoC moderno envolvem CPUs *multicore*, GPUs capazes de realizar processamento de propósito geral (também conhecido por GPGPU), além de elementos coprocessadores como o NEON dos ARMs modernos.

Alguns SoCs possuem até mesmo um processador digital de sinais (DSP) agregado, o que permite acelerar ainda mais aplicações que envolvem transformadas de Fourier, multiplicações matriciais, dentre outros tipos de operações tais que são de fundamental importância em rotinas de processamento de imagens e visão computacional.

Configurar um ambiente de programação e desenvolvimento, conhecer e saber programar essas arquiteturas e utilizar os benefícios que elas trazem, em suma, saber programar sistemas que tirem o máximo de seus recursos, é algo ainda difícil, penoso e incipiante.

1.2. Contribuição

As principais contribuições desse trabalho que podem ser destacadas são:

- Apresentação de como rotinas de Processamento de Imagens podem ser computadas em GPUs embarcadas por meio de OpenGL ES 2.0.
- Demonstração de como trabalhar com OpenGL ES 2.0 em sistemas embarcados com Linux.
- Execução de convolução com Filtro de Sobel em GPU embarcada com OpenGL ES 2.0 e comparativo da sua execução em CPU com OpenCV.
- Avaliar a viabilidade de uso de sistemas embarcados com GPU na realização de tarefas de processamento de imagens.

Esse trabalho contribui para a área por apresentar uma alternativa à tradicional abordagem de computar rotinas de processamento de imagens em CPUs embarcadas, ao passo que chama a atenção para o uso de recursos de GPU por meio de OpenGL ES 2.0, o que torna os exemplos demonstrados e aplicados no trabalho compatíveis com uma boa gama de dipositivos presentes no mercado.

De outra forma, no decorrer da pesquisa realizada também foi constatado que há pouca documentação e poucos exemplos realmente funcionais que demonstrem a técnica em operação.

Sendo assim, pelo fato de o trabalho também apresentar códigos, estruturas e exemplos bem documentados e funcionais também se torna uma contribuição importante para embasar o desenvolvimento de interessados, apresentando o caminho de como fazer uso de GPUs embarcadas para conseguir um acréscimo no desempenho de recursos de processamento de imagens.

1.3. Objetivo

O objetivo deste trabalho consiste na proposta de utilizar GPUs embarcadas para acelerar a execução de algoritmos de processamento de imagens por meio de rotinas com OpenGL ES 2.0, juntamente com a aplicação de um *benchmark* para comparar analiticamente o ganho de desempenho frente a execução em CPU embarcarda.

O algoritmo-referência para ambas as aplicações, com execução em GPU embarcada e CPU, é o Filtro de Sobel.

1.4. Organização do Trabalho

Para melhor distribuir e organizar o conteúdo do trabalho, o mesmo foi dividido em 6 capítulos, a saber:

- **1. Introdução** – Presente capítulo, que trata de apresentar a introdução histórica do tema abordado, motivação, contribuição, objetivos, terminando com a apresentação e discussão de trabalhos publicados com relação ao tema.
- **2. História –** Apresentação do fluxo histórico de tecnologias que contribuíram para o atual estado da arte das ferramentas utilizadas como base, princípio e motivação para a proposta de aceleração de desempenho de algoritmos de visão computacional em sistemas embarcados.
- **3. Fundamentação Teórica** – Capítulo que trata de abordar todos os conceitos que fundamentam a realização do presente trabalho, começando por um estudo de trabalhos relacionados, e depois apresentando as tecnologias e plataformas necessárias para implementar, testar e executar a otimização proposta em sistemas embarcados para processamento de imagens.
- **3. Materiais e Métodos** – Neste capítulo serão apresentadas as ferramentas computacionais e os sistemas embarcados utilizados como base para o desenvolvimento do trabalho, além das técnicas empregadas para testes, programação, execução e comparação de resultados.
- **4. Resultados** – Com o emprego das técnicas destacadas no capítulo de Materiais e Métodos na plataforma trabalhada, serão apresentados neste capítulo os resultados obtidos dos testes realizados.
- **5. Conclusões** – Por fim, apresentam-se neste capítulo conclusões a respeito da viabilidade da implementação de sistemas que utilizem os recursos conjuntos de sistemas embarcados modernos para processamento de imagens.
- **6. Trabalhos Futuros** – Com os resultados obtidos no trabalho, tanto a título de execução de rotinas como a título do desenvolvimento dos códigos-fontes das aplicações, neste capítulo são apresentados os planos para a continuidade do trabalho.

2. História

O que começou em meados de 1980 como uma plataforma alternativa aos caríssimos computadores da época, veio a se tornar uma arquitetura líder no mercado atual, graças à iniciativa de projetar processadores com arquitetura RISC frente aos processadores Intel CISC da época. Este processo deu início ao processador Acorn Risc Machine feito pela empresa Acorn, que passou a se chamar ARM em 1990, cujo desenvolvimento contou com apoio da *Apple*, VLSI e da própria Acorn. Além disso, o objetivo da empresa ARM é, e sempre foi, o de criar arquiteturas e projetar sistemas confiáveis, com bom desempenho e baixo consumo de energia. Em suma, a ARM licencia seus projetos e trabalha com parceiros e associados na criação de projetos específicos, deixando a cargo de seus clientes a tarefa de fabricar e vender os *chips*. Exemplos de parceiros e associados ARM são Samsung, NXL, Freescale, dentre outros (LANGBRIDGE, 2014).

Já em meados de 1991, Linus Torvalds criou o sistema operacional Linux, de código aberto, quando estudava na Universidade de Helsinki. O que o motivou a trabalhar em um sistema operacional aberto foi uma série de desavenças com o criador do Minix, Andrew Tanenbaum, que recusara pedidos de melhorias feitos por Torvalds. Insatisfeito, trabalhou em seu próprio sistema, fazendo o anúncio ao grupo "comp.os.minix", já traduzido para o português:

Estou fazendo um (gratuito) sistema operacional (apenas um hobby, não será grande e profissional como o GNU) para clones do AT 386 (486). Isso tem crescido desde abril, e está começando a ficar pronto. Eu gostaria de receber qualquer retorno em coisas que as pessoas gostam ou não gostam do MINIX, pois meu OS relembra ele de alguma forma (mesmo layout físico do sistema de arquivos (devido a razões práticas) além de outras coisas.[12]

Mal imaginava o estudante Linus Torvalds que anos mais tarde o seu sistema seria amplamente utilizado em todo o mundo. Como exemplo, o sistema operacional Linux é utilizado por 494 dos 500 computadores mais rápidos do planeta, como observado na listagem de novembro de 2015 feita pelo Top500[13].

O fato de ter sido liberado como sistema aberto permitiu que mais e mais programadores tivessem acesso aos códigos-fontes do sistema, e realizassem tanto modificações,

---

[12] Maiores detalhes a respeito da história do sistema Linux, e conversas trocadas entre Linus Torvalds e demais usuários podem ser vistos nesse link aqui, em inglês: https://www.cs.cmu.edu/~awb/linux.history.html

[13] Dados obtidos do *site* TOP500, que reúne dados dos 500 computadores mais rápidos do planeta. Detalhes a respeito dos sistemas operacionais da lista de computadores pode ser visto por completo neste *link*: http://www.top500.org/statistics/details/osfam/1.

correções e melhorias no sistema. Em 1995, Bruce Perens cria a suíte Busybox, agregando uma coleção de utilitários em linha de comando, tornando possível colocar um instalador do sistema Debian em um disquete de 1.44 MB. O desenvolvimento do Busybox foi um grande passo para tornar possível a execução do sistema Linux em dispositivos portáteis com menor poder de processamento (SIMMONDS, 2013).

E além de modificações diversas, desenvolvedores e projetistas poderiam, por terem acesso aos códigos-fontes do sistema Linux, portar o sistema para outras plataformas. Foi assim que no verão de 1994 surgiu a primeira versão do sistema Linux para plataforma ARM, começando como uma versão do Kernel 1.0 do Linux para a máquina Acorn A5000[14], mostrada na Figura 6.

Figura 6 - Acorn A5000. Fonte: www.anytux.org.

Paralelamente, outros desenvolvedores e projetistas portavam o sistema Linux para outras plataformas tais como MIPS, m68k, dentre outras.

Três anos depois, em 1997, Dave Cinege utilizou o Busybox para criar uma distribuição Linux em um disquete, capaz de tornar um computador em um roteador. E em 1998 David e Retkowski desenvolvem o primeiro roteador sem fio utilizando Linux (SIMMONDS, 2013).

Os primeiros produtos, ou melhor, sistemas embarcados baseados no sistema Linux, começam a aparecer já em 1999, tais como o sistema interativo de televisão TiVo, mostrado na Figura 7, sistema de câmeras AXIS 2100, dentre outros (SIMMONDS, 2013).

[14] Dados obtidos no site ARM Linux. Maiores detalhes sobre a história do Linux em sistemas ARM pode ser visto por completo neste *link*: http://www.arm.linux.org.uk/docs/history.php.

Figura 7 - Primeiro TiVO. Fonte: codinghorror.com.

No mesmo ano de 1999, a suíte de visão computacional OpenCV foi oficialmente lançada pela Intel, com o objetivo de promover avanços e disseminação de conhecimento na área de visão computacional, além de permitir o desenvolvimento de avançadas aplicações comerciais por meio da disponibilização de uma suíte de código aberta (SIMMONDS, 2013).

E nos anos de 2000 a 2005, começam a aparecer no mercado dispositivos móveis utilizando o sistema Linux, tais como Sharp Zaurus e Nokia 770, mostrados na Figura 8. Ambos aparelhos foram desenvolvidos para servirem como PDAs, ou, traduzido para o português, Assistentes Pessoais Digitais. O aparelho Nokia 770, com sua tela maior, ficou conhecido como *internet tablet* (SIMMONDS, 2013).

Figura 8 - Sharp Zaurus à esquerda, e Nokia 770, à direita. Fontes: Sharp.com e Nokia.com.

Já em 2006, inspirados pelo Acorn BBC Micro de 1981, Eben Upton e um grupo de professores e acadêmicos decidem criar um computador com tamanho reduzido e preço acessível para inspirar crianças e entusiastas de computação, o que foi o início da Fundação Raspberry PI[15]. O foco da Fundação era criar dois modelos de microcomputadores, custando $ 25 e $ 35 cada. Eben Upton é engenheiro da Broadcom, e fazendo uso da sua experiência como projetista de *hardware* juntamente com o apoio de seus colegas, conseguiram em pouco tempo apresentar o protótipo da Raspberry Pi, mostrado na Figura 9, demonstrando a viabilidade do projeto.

[15] Detalhes sobre a história da Fundação Raspberry Pi podem ser encontrados neste *link*: http://www.raspberrypi.org/about

Figura 9 - Uma das primeiras versões de testes da Raspberry PI. Fonte: http://elinux.org/.

A Raspberry Pi acabou sendo uma alavanca aceleradora para o surgimento de novas comunidades, *kits* didáticos e plataformas de desenvolvimento para sistemas Linux embarcados. Haviam plataformas de desenvolvimento anteriores ao Raspberry Pi, mas custando bem mais que seus $ 35. Veja na Figura 10 alguns exemplos de placas voltadas ao aprendizado e desenvolvimento com Linux Embarcado, além da própria Raspberry Pi.

Figura 10 - Em ordem da frente para trás: Raspberry PI, MK802III, BeagleBone Black, PCDuino, CubieBoard, Odroid-U2. Fonet: www.open-electronics.org

Com a ampla disseminação de dispositivos móveis em todas as esferas do mercado, cresce então a demanda e a competição por dispositivos que ofereçam mais funcionalidades e mais atrativos ao público consumidor. Uma das abordagens empregadas é o uso dos recursos oferecidos por plataformas modernas, tais como processadores *multicore* e processamento de imagens, para permitir o controle e a interação com computadores e dispositivos eletrônicos não mais usando teclado ou mouse, mas por meio de gestos, como mostrado na Figura 11, que destaca essa funcionalidade já presente em alguns modelos de televisões e celulares.

Figura 11 - Televisão SmartTV da Samsung com câmera e controle via gestos e Celular Smartphone Samsung Galaxy S4, que possui controle por gestos. Fonte: Samsung.com.

Com relação ao sistema Android, a partir da sua versão 4.0[16] os aparelhos possuem recurso de desbloqueio por meio de reconhecimento facial, chamado na plataforma por *Face Unlock*. Exemplo da aplicação em execução é mostrado na Figura 12.

Figura 12 - Celular Android com reconhecimento de face. Fonte: talkandroid.com.

Já a Microsoft disponibiliza a aplicação *HERE City Lens*[17] para seus aparelhos *smartphones* Lumia. Um exemplo da aplicação é mostrado na Figura 13. A aplicação faz uso de Realidade Aumentada, somado a recursos de geolocalização para fornecer um panorama ao usuário de locais à sua volta, como restaurantes, bares, etc.

Figura 13 - Nokia Lumia com recurso de realidade aumentada no programa Here City Lens. Fonte: gigaom.com.

Agregando ainda mais funcionalidades às câmeras presentes em aparelhos *smartphones*, a biblioteca ZXing[18] torna possível a leitura de códigos de barras tradicionais. Tal recurso permitiu

[16] Anúncio de novidades no Android *Ice Cream Sandwich*, versão 4.0, disponível em: http://www.android.com/about/ice-cream-sandwich/
[17] Maiores detalhes a respeito do *Nokia City Lens* podem ser vistos no site do projeto, disponível em: http://www.nokia.com/global/apps/app/here-city-lens/
[18] Página do projeto: https://code.google.com/p/zxing/

o desenvolvimento de aplicações com capacidade de identificação de produtos ou pagamento de boletos, como mostrado na Figura 14.

Figura 14 - Celular Android fazendo leitura de código de barras. Fonte: happyandroid.com

Todas essas funcionalidades só são possíveis graças a novos e modernos processadores embarcados, os famosos SoCs[19], que agregam não somente CPUs *multicore*, mas placas gráficas (GPUs) e coprocessadores capazes de auxiliar nas pesadas tarefas realizadas pelos algoritmos de visão computacional empregados no desempenho dos recursos necessários.

[19] SoC é a abreviação em inglês de *System-on-a-chip*, ou, em português, de sistema em um *chip*. Ao contrário dos microcontroladores, que são considerados *computadores em um chip*, os SoCs reúnem elementos de um sistema inteiro, tais como unidade de radiofrequência, GPU, CPU, controlador de *display*, dentre outros. Maiores detalhes sobre o termo podem ser vistos neste *link*: http://whatis.techtarget.com/definition/system-on-a-chip-SoC

3. Fundamentação Teórica

O presente capítulo irá discutir o embasamento teórico necessário para a compreensão e o desenvolvimento do trabalho, começando por apresentar trabalhos relacionados que tratam de análise de desempenho computacional e de algoritmos de visão computacional em sistemas embarcados, como também apresenta os conceitos relacionados às áreas de *hardware* pertinentes às plataformas ARMs e seus componentes, sistemas Linux, OpenCV, OpenGL e demais meios que auxiliam no uso otimizado de recursos de máquina.

3.1. Trabalhos Relacionados

O presente capítulo irá discutir e abordar artigos, teses, notícias e demais documentos que tratam de trabalhos realizados na área de otimização de recursos de dispositivos móveis, com destaque para dispositivos com processadores *multicore*, além de sistemas com GPU integrada e coprocessadores aritméticos.

Pulli et. al. (2012) avaliam o desempenho de tempo real em visão computacional com a biblioteca OpenCV. Os autores do artigo são Kari Pulli, pesquisador da NVIDIA, uma empresa multinacional famosa por suas poderosas placas de vídeo, e os outros são Anatoly Baksheev, Kirill Kornyakov e Victor Eruhimov, pesquisadores da empresa Itseez, uma empresa especializada em desenvolvimento de algoritmos de visão computacional para dispositivos móveis tais como *smartphones*, *tablets* e demais sistemas embarcados.

No artigo de Pulli et. al. (2012), os autores novamente ressaltam que as tarefas de visão computacional são computacionalmente pesadas. Com a experiência que possuem na área, destacam que muitos cenários de visão computacional em tempo real requerem que o processamento de um único quadro seja feito em uma janela de 30 a 40 milissegundos. Esse requerimento é muito desafiador, ainda mais tendo em mente as limitações computacionais de dispositivos embarcados.

Com base nesse desafio, Pulli et. al. (2012) conduziram um estudo a cerca do aumento de desempenho para visão computacional bom base no uso de recursos de GPU, em conjunto com CPU, em se tratando de plataformas modernas na NVIDIA que permitem a execução de código na GPU, as chamadas GPGPU. Como destacam, as GPUs não são tão flexíveis em operações como as CPUs, mas desempenham tarefas paralelas de modo muito mais eficiente, e um número cada vez mais crescente de aplicações, mesmo não gráficas, estão sendo reescritas usando linguagens GPGPU.

Desde 2010, a biblioteca OpenCV possui rotinas para realizar processamento de imagens na GPU, mas é algo que somente é suportado pela plataforma CUDA, da NVIDIA (CUDA, 2016). Outras plataformas, ou APIs, que permitem a execução de código na GPU tal como a OpenCL ainda carece de maior suporte por parte do OpenCV.

Os autores realizaram um *benchmark* para avaliar o ganho de desempenho entre rotinas de visão computacional com relação à execução na CPU e na GPU. O *benchmark* foi executado em uma CPU Intel Core i5-760 2.8 GHz com uma placa gráfica NVIDIA 580 GTX, com a biblioteca OpenCV otimizada para Intel SSE (*Steaming SIMD Extensions*) e TBB (*Threading Building Blocks*) para suporte a processamento *multicore*, porém os autores destacam que nem todos os algoritmos da biblioteca utilizam as otimizações. Os resultados obtidos pela execução de algumas rotinas são mostrados na Figura 15, destacando que em algumas rotinas mais simples o desempenho da GPU é cerca de 30 vezes mais rápido, e em outras rotinas, mais complexas, cerca de 10 vezes mais rápida.

Infelizmente, os autores apenas apresentaram resultados, compararam os dados obtidos e discutiram possíveis abordagens para melhorias de desempenho. Não foram fornecidos demais detalhes pertinentes à codificação de programas.

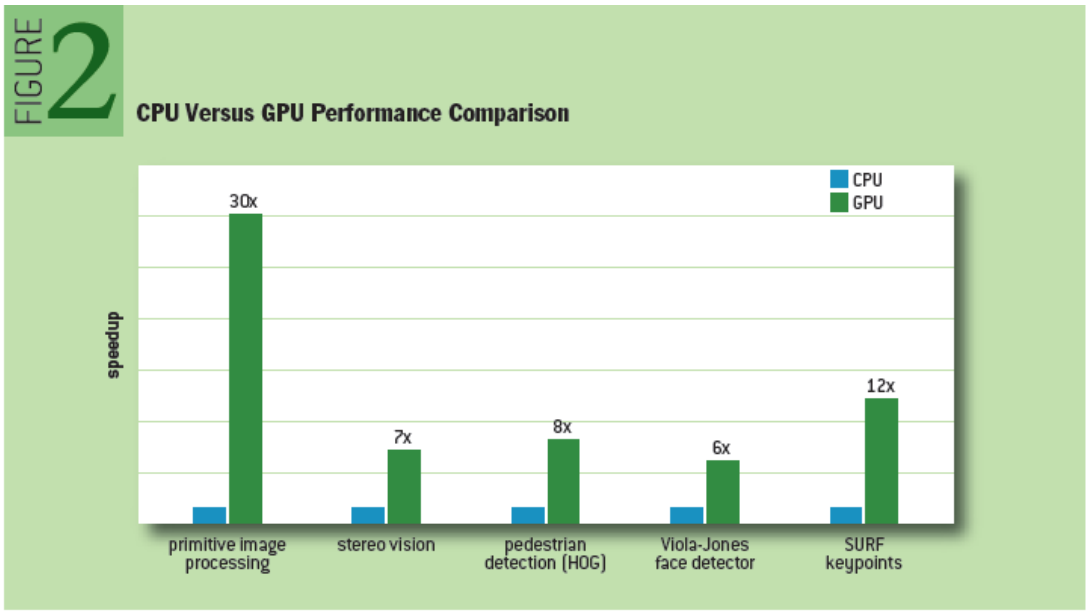


Figura 15 - Comparativo entre desempenhos de performance em Visão Computacional entre CPU e GPU. Fonte: PULLI et. al. (2012).

Depois de realizar um panorama a respeito dos desafios de processamento assíncrono entre CPUs e GPUs, Pulli et. al. (2012) passam para os desafios da visão computacional em dispositivos móveis, que possuem SoCs (*System on Chip*) que integram componentes como CPU e GPU. O foco nesta parte do trabalho publicado foi na plataforma Tegra 3, um ARM Cortex-A9 com quatro núcleos dotado de um coprocessador NEON. O coprocessador NEON é similar ao

SSE da família Intel, pois realiza processamento SIMD (*Single Instruction Multiple Data*). Dependendo da arquitetura implementada pelo fabricante, o coprocessador NEON pode processar de 8 a 16 pixels de uma vez, podendo resultar em um aumento médio de 3 a 4 vezes no tempo de processamento de rotinas de processamento de imagens.

Para comprovar as vantagens do coprocessador NEON, Pulli et. al.(2012) conduziram uma outra sequência de testes para comparar o desempenho entre a execução de rotinas na CPU ARM Cortex A9, e no coprocessador NEON do SoC Tegra 3. Para o *benchmark* na plataforma Tegra 3, os autores compararam o ganho de desempenho da biblioteca OpenCV original, e da biblioteca otimizada para NEON. Como esperado, houve um ganho de desempenho decorrente do fato de o coprocessador NEON realizar instruções SIMD, ideais para acelerar operações aritméticas vetoriais. Os resultados dessa comparação são mostrados na Figura 16.

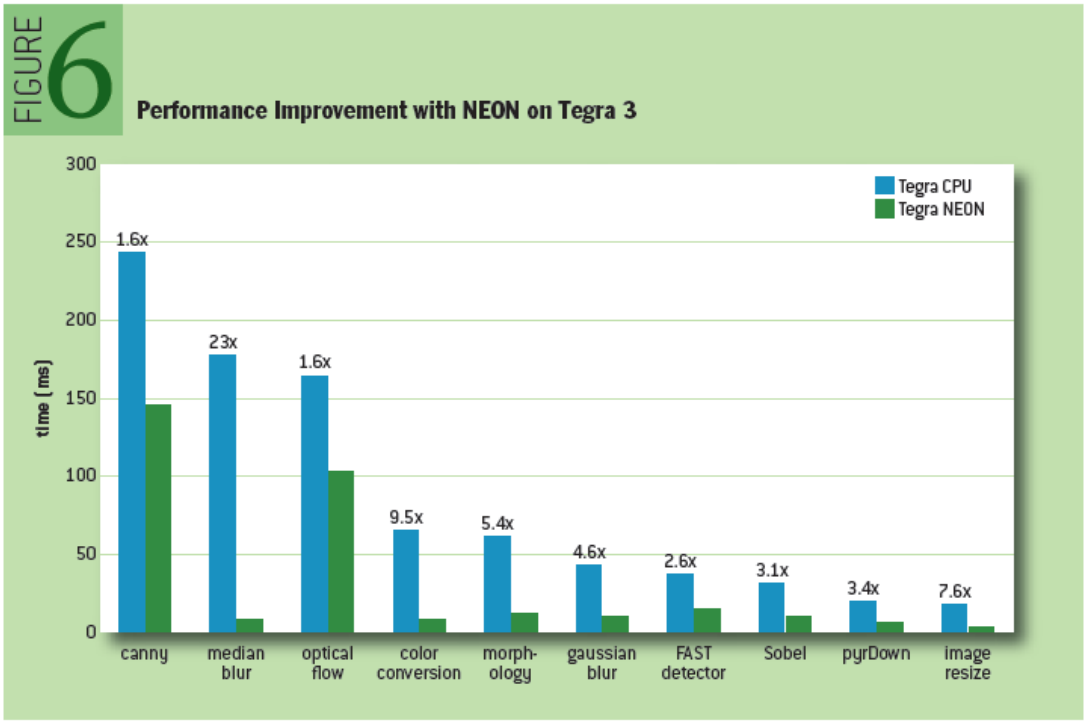


Figura 16 - Gráfico de comparativos de desempenho entre execução de rotinas de visão computacional na CPU e no Coprocessador NEON do SoC Tegra 3. Fonte: PULLI et. al. (2012).

Além disso, os autores também abordam o uso de TBB para otimizar o uso de múltiplas CPUs, de modo a permitir a execução de diferentes *threads* da aplicação, deixando a cargo do sistema operacional o controle de carga de cada núcleo de processamento. Uma abordagem recomendada no artigo é a de dividir tarefas de baixo nível em várias subtarefas, que produzem

resultados mais rápidos, e uma *thread*[20] rodando em *background*[21] para obter e agrupar os resultados quando estiverem concluídos, enquanto que rotinas do programa principal trabalham em outras partes do problema.

Concluindo, PULLI et. al. (2012) ressaltam que as operações *multithreads*[22] estão sujeitas ao número de núcleos e à velocidade de acesso à memória, e que tarefas *multithreads* podem ser combinadas com rotinas NEON, mas tendo em vista a análise do quanto essas tarefas irão ocupar do barramento de memória durante o processo. Infelizmente, no ano de publicação do artigo, ainda não haviam no mercado SoCs ARM com unidades GPU capazes de executar processamento de propósito geral, tais como os permitidos pelas plataformas CUDA e OpenCL, e no presente momento, principalmente no que tange ao OpenCL, já existem diversos SoCs ARM com GPUs compatíveis com as especificações OpenCL, o que aumenta ainda mais a gama de possibilidades de operações para visão computacional. Além disso, os autores do artigo também afirmam que há ainda muito trabalho a ser feito, em otimizações de rotinas de processamento de imagens e visão computacional em sistemas embarcados, dado que poucas rotinas da biblioteca OpenCV já vêm otimizadas para utilizar recursos como NEON, caso disponíveis no dispositivo.

Já com relação ao uso de rotinas para processamento paralelo, há então a possibilidade de utilizar OpenCL em equipamentos que suportem essa especificação, o que permite a execução de código em paralelo nos vários núcleos de uma GPU. De outra forma, podemos também utilizar APIs como *Threading Building Blocks* (TBB) e *Open Multi-Processing* (OpenMP), que permitem a execução de código em paralelo em núcleos de uma CPU *multicore*, por exemplo.

Dentre as publicações pesquisadas que abordam os temas de processamento paralelo, foram encontrados o artigo de Thouti e Sathe (2012), que aborda um comparativo entre o desempenho de OpenMP contra OpenCL, o artigo de Maghazeh et. al. (2013), que avalia um comparativo de desempenho entre GPU embarcada e GPU *desktop*, a dissertação final de mestrado de Ali (2013), aonde é feito um estudo comparativo de modelos de programação paralela para computação *multicore*, em conjunto com o artigo de Tristam e Bradshaw (2010), em que os autores avaliam a performance e características de código de três modelos de programação paralela em C++.

No texto de Thouti e Sathe (2012), os autores fazem um estudo acerca do desempenho de implementações paralelas de alguns algoritmos em execução numa máquina com 2 processadores

---

[20] *Thread* é um termo em inglês que denota uma tarefa executada por um processador (CPU).
[21] Nesse contexto, *background* significa dizer que a tarefa (thread) é executada fora do contexto principal da aplicação, ou seja, "ao fundo".
[22] *Multithreads* é a indicação para execução de rotinas capazes de executar várias tarefas (*threads*) em um processador (CPU).

Intel Xeon com frequência de 2.67 GHz, com um total de 24 *threads* possíveis de execução, em conjunto com uma placa gráfica NVIDIA Quadro FX 3800, modelo que possui 192 núcleos de processamento e 1 GB de memória dedicada. Para execução de código em paralelo na CPU, optaram pela API OpenMP, por ser uma biblioteca popular para programação de computadores *multicore* com memória compartilhada. Já a escolha pela biblioteca OpenCL, e não por CUDA, se deu pelo fato de OpenCL ser um modelo de programação multiplataforma, cuja especificação e padronização é mantida pelo *Kronos Group*, enquanto que CUDA é específico para plataformas NVIDIA. O sistema operacional utilizado nos testes foi o Fedora X86_64, com Kernel Linux versão 2.6, e versão do compilador GCC, 4.6.

Para efeitos de *benchmark*, Thouti e Sathe (2012) conduziram uma bateria de testes de programas com implementação paralela, de multiplicação de matrizes com grandes dimensões. Eles escolheram testar multiplicação de matrizes por ser uma tarefa amplamente utilizada em processamento de imagens, e por envolver tarefas computacionais mais facilmente paralelizáveis. Feito isso, mediram o tempo gasto na execução dos algoritmos com implementação sequencial, com uso da biblioteca OpenMP e com uso de OpenCL. Os resultados obtidos são mostrados no gráfico da Figura 17.

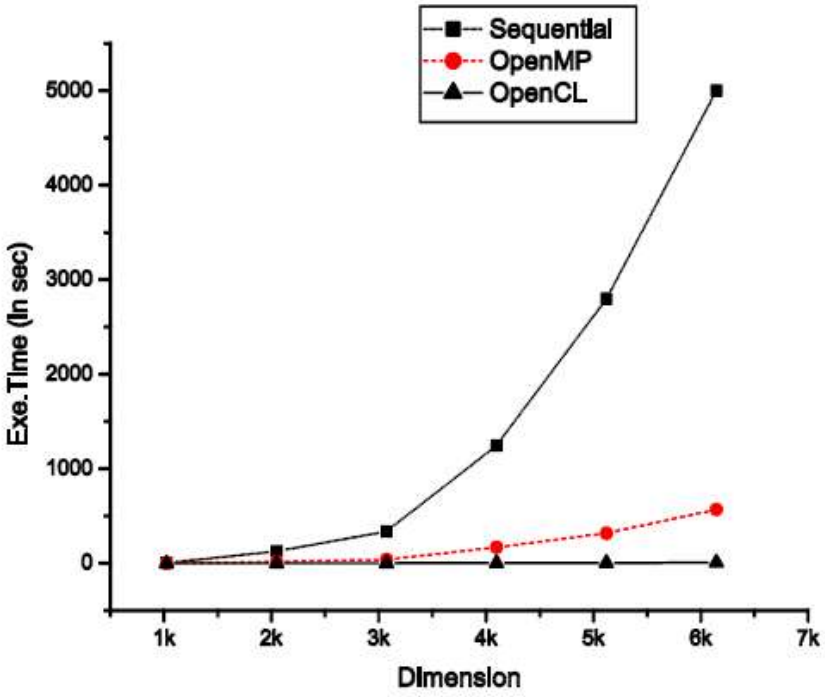


Figura 17 - Tempo de execução por dimensão da matrix. Fonte: Thouti e Sathe (2012).

Como observado pelos resultados obtidos por Thouti e Sathe (2012), o tempo de execução para o programa implementado em OpenCL foi quase linear, mesmo envolvendo multiplicação matricial com matrizes de ordem 6000. Já a implementação sequencial apresentou aumento de

tempo exponencialmente proporcional à dimensão da matriz, enquanto que a implementação OpenMP começou a tomar mais tempo que a implementação OpenCL com matrizes de dimensão superior a 1024. A divergência de desempenho entre OpenMP e OpenCL pode ser melhor observada no gráfico mostrado na Figura 18, na qual Thouti e Sathe (2012) fizeram uma análise comparativa entre o aumento gradativo de tempo gasto para processar matrizes com dimensões maiores. Como observado, a implementação com OpenCL mantém um tempo inalterado de execução, enquanto que a mesma com OpenMP apresenta aumento de tempo com aumento da dimensão das matrizes. Vale lembrar que a plataforma utilizada no trabalho foi uma NVIDIA Quadro FX 3800, com 192 núcleos de execução.

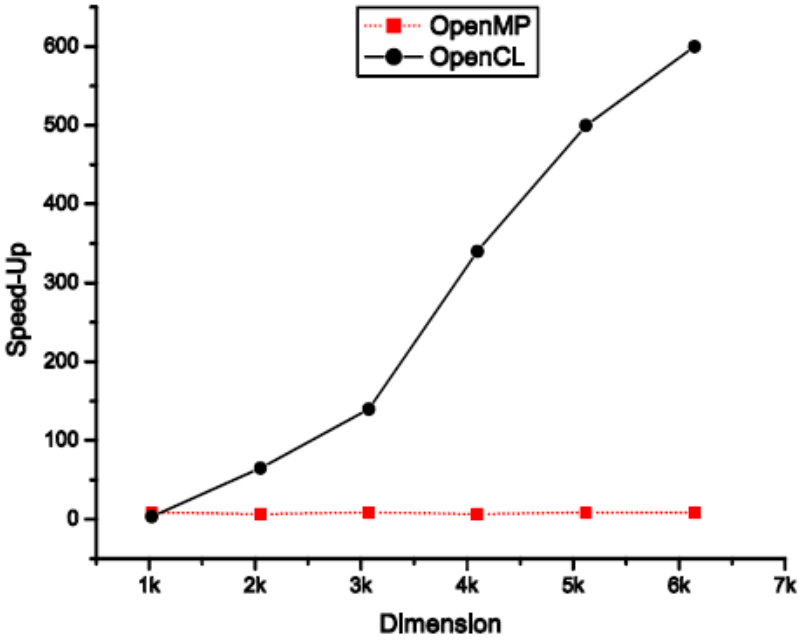


Figura 18 - Fator de aceleração entre OpenMP e OpenCL. Fonte: Thouti e Sathe (2012).

No final do artigo, Thouti e Sathe (2012) destacam que trabalhar com OpenCL envolve uma boa carga de trabalho, como alocação de memória, configurações de *kernel* de execução, carregamento e obtenção de dados da GPU, dentre outros parâmetros que agregam *overhead* de tempo em uma aplicação OpenCL. Porém, mesmo com esse *overhead*, o desempenho de OpenCL em processamento paralelo de grandes massas de dados é superior aos demais. Dessa forma, a conclusão do trabalho é que, em ordem de desempenho, um algoritmo paralelo implementado em OpenCL possui um desempenho maior que o mesmo implementado em OpenMP, e a implementação sequencial é a mais lenta.

Já o trabalho de Tristam e Bradshaw (2010) envolveu a comparação de desempenho computacional em processamento paralelo envolvendo o uso das bibliotecas OpenMP e TBB, e também o modelo de implementação mais “pura”, fazendo uso de *pthreads*. Outra técnica que os

autores utilizaram foi a de implementar *Boost.Threads*, que nada mais é do que uma outra técnica de implementação em baixo nível de *threads*.

Para realizar a comparação, o algoritmo escolhido pelos autores foi o cálculo do fractal de Mandelbrot, e todos os testes foram executados em uma máquina Intel Core 2 Quad Q9400, com quatro núcleos a 2.66 GHz, 4 GB de memória RAM DDR2 800 MHz, com sistema operacional Gentoo Linux x64. As versões das aplicações utilizadas foram OpenMP 3.0, TBB 2.2 Update 1, e GCC 4.4.3. Para melhorar a consistência dos testes, os autores desabilitaram todos os serviços e aplicações não-essenciais aos objetivos dos testes, incluindo a interface gráfica.

Os testes realizados por Tristam e Bradshaw (2010) foram executados com e sem a otimização para instruções SSE, no processamento do algoritmo do fractal de Mandelbrot para resoluções de 16000, 20000, 240000, 28000 e 32000. O uso de otimizações SSE é um fator chave, por englobar um conjunto de instruções SIMD com foco em operações aritméticas.

Cada um dos testes de Tristam e Bradshaw (2010) foi repetido 5 vezes, e os resultados finais obtidos são a média das execuções. Para verificar ainda mais qualquer eventual ganho de desempenho por meio do compilador utilizado, os autores compilaram os programas utilizando compilador G++, da suíte GNU, e ICC, que é o Compilador C da Intel. Os resultados obtidos pela bateria de testes para validação de ganho entre os compiladores G++ e ICC, com e sem otimização SSE para implementações sequenciais e paralelas do algoritmo de Mandelbrot pode ser visto no gráfico da Figura 19.

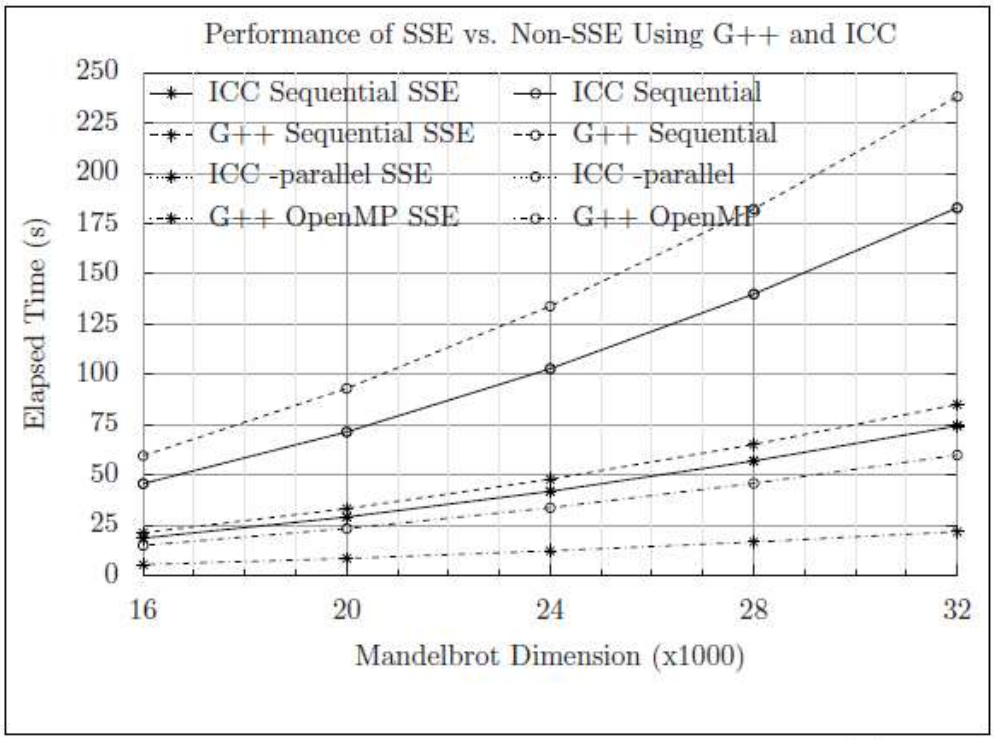


Figura 19 - Comparativo de desempenho entre algoritmo compilado com G++ e ICC, com e sem otimizações SSE. Fonte: Tristam e Bradshaw (2010).

Pelo que pode ser observado na Figura 19, a implementação do algoritmo de Mandelbrot com a biblioteca OpenMP, compilado com G++ sem a otimização SSE, possui quase o mesmo desempenho do que a implementação sequencial, compilado com ICC e com a otimização SSE.

Já na Figura 20, é possível observar o gráfico com dados resultantes dos testes nas implementações do algoritmo de Mandelbrot compiladas com ICC e com G++, fazendo uso de rotinas paralelas com *Boost.Thread*, *pthreads*, OpenMP e TBB, respectivamente.

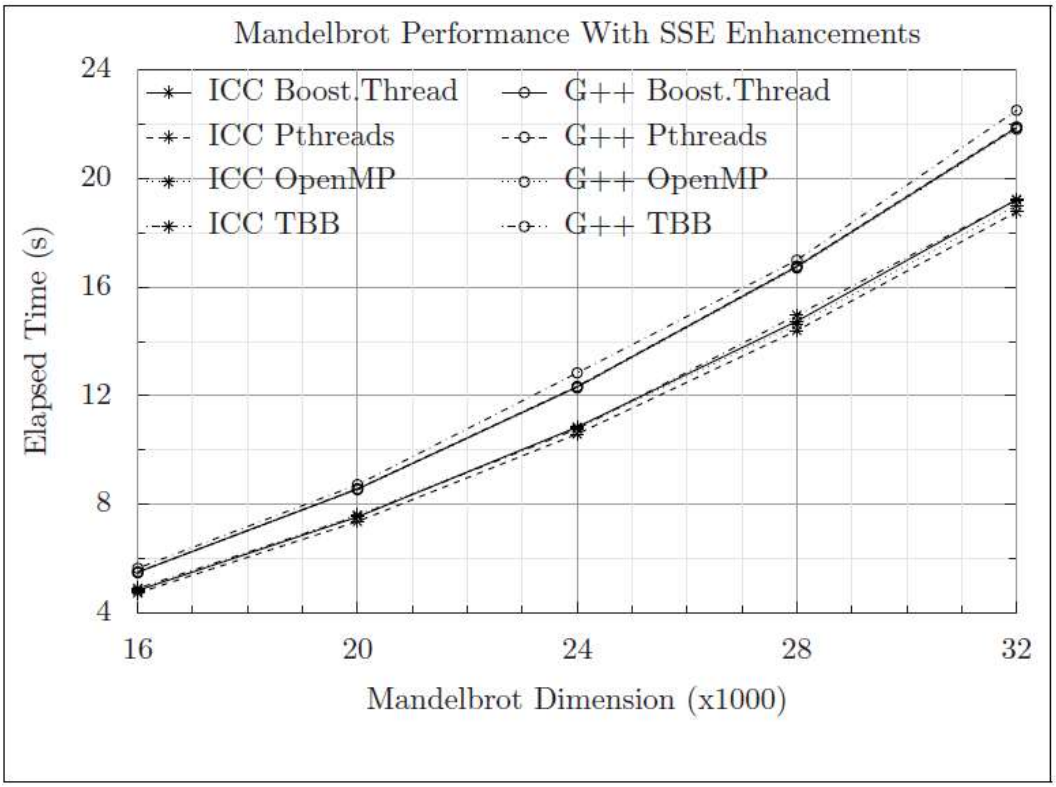


Figura 20 - Gráfico de tempo gasto com implementações paralelas com otimização SSE habilitada. Fonte: Tristam e Bradshaw (2010).

Como pode ser observado no gráfico, a execução do algoritmo compilado pelo ICC (correspondente às linhas inferiores) apresenta um desempenho superior ao compilado pelo G++ (correspondente às linhas superiores). Pelo gráfico, também se conclui que a execução com *pthreads* é a mais rápida, seguida pelos demais, que apresentam comportamento bem similar. Para facilitar a compreensão do desempenho das implementações com o aumento da complexidade do algoritmo, Tristam e Bradshaw (2010) sintetizaram o gráfico mostrado na Figura 21, que mostra o ganho de desempenho para cada implementação, ao longo do aumento da dimensão do algoritmo de Mandelbrot.

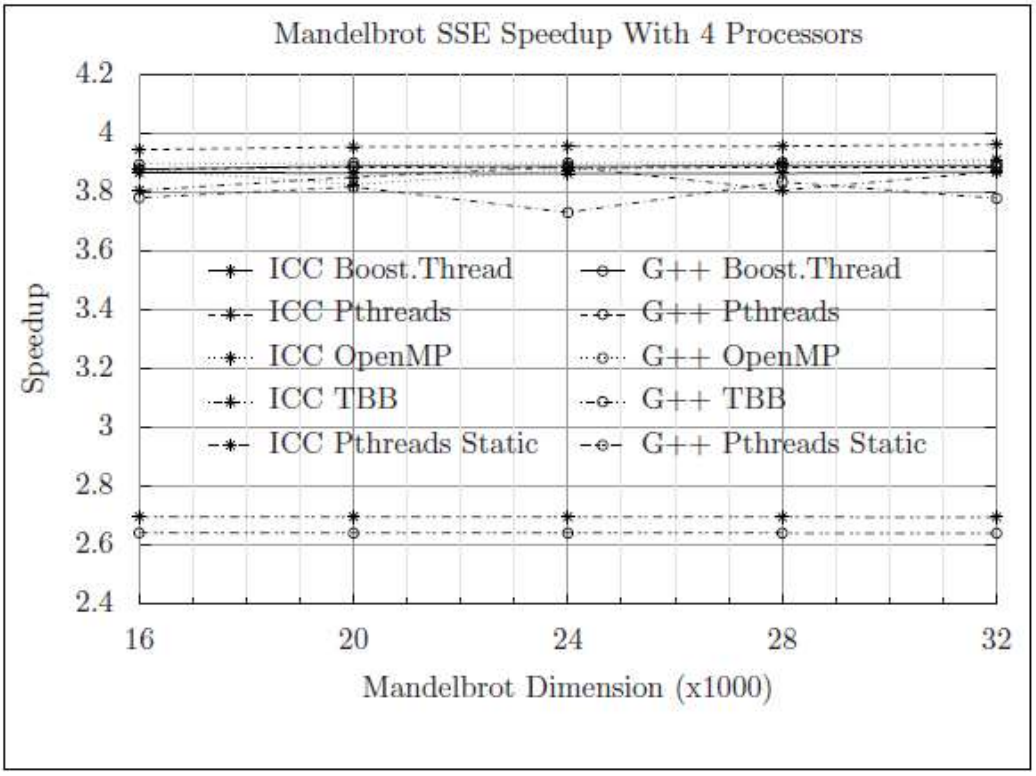


Figura 21 - Ganho de desempenho das implementações paralelas com otimização SSE. Fonte: Tristam e Bradshaw (2010).

Para o caso do algoritmo de Mandelbrot, a implementação paralela utilizando *pthreads* compilada com ICC apresentou o melhor desempenho, seguida por OpenMP e *Boost.Thread*. Os autores discorrem que OpenMP apresentou um melhor desempenho que TBB neste caso, em decorrência de sua natureza voltada para processamento de dados em paralelo, enquanto que TBB é projetado para um nível maior de abstração, aonde tarefas são paralelizadas em um meio orientado a objetos. Por fim, os autores montaram a Tabela 1 com as características dos códigos utilizados nos testes, envolvendo cada tipo de implementação, quantas linhas de código a implementação tomou, e quantas linhas foram necessárias para alterar a implementação sequencial padrão para a implementação otimizada.

Tabela 1 - Tabela com características de código para as implementações paralelas de Mandelbrot. Fonte: Tristam e Bradshaw (2010)

Implementation	Lines	Modified
Sequential	77	-
SSE Sequential	93	38
Boost Static	119	48
Boost Dynamic	210	139
OpenMP	78	1
Pthreads Static	125	57
Pthreads Dynamic	202	134
Thread Building Blocks	101	34

Por fim, Tristam e Bradshaw (2010) concluem que a biblioteca mais simples para ser utilizada em uma implementação paralela é a OpenMP, cuja implementação tomou apenas 1 linha de código a ser modificada, e traz um nível bom e aceitável de performance em aplicações paralelas. Já a biblioteca TBB demanda um pouco mais de esforço para ser implementada, mas é mais flexível, dada sua abstração de tarefas paralelas. E apesar de *pthreads* e *Boost.Threads* terem alcançado maior desempenho, tomam mais tempo para ser implementadas, requerem maior experiência do usuário, pois como a programação das rotinas ocorre em um nível mais baixo do que nas bibliotecas OpenMP e TBB, o controle do paralelismo das *threads* fica na mão do programador.

Já o artigo de Maghazeh et. al (2013) trata de comparar o desempenho de uma GPU embarcada contra o desempenho de uma GPU *desktop*, passando por uma análise arquitetural, e finalizando com um *benchmark* de três aplicações em OpenCL, colocadas para execução nas GPUs.

Os sistemas escolhidos por Maghazeh et. al. (2013) foram a plataforma i.MX6 Sabre Lite, que possui um ARM Cortex-A9 com quatro núcleos a uma frequência de 1,2 GHz, e uma GPU Vivante GC2000, com quatro núcleos SIMD, a uma frequência de 500 MHz. Os autores destacam que a GPU Vivante GC2000 possui um *cache* de memória de instruções capaz de acomodar um total de 512 instruções, e um *cache* de memória total de 4 KB. Enfim, as especificações técnicas dessa GPU são focadas em desempenho para baixo consumo de energia, o que não é o caso da GPU *desktop* NVIDIA Tesla M2050, que possui um total de 448 unidades de processamento a uma frequência de 1147 MHz. A estação *desktop* associada à GPU NVIDIA também continha 2 processadores Intel Xeon E5520, com 8 núcleos. Com relação ao sistema operacional escolhido para os testes, os autores apenas destacam o sistema do SoC i.MX6, que foi o sistema Linux com Kernel 2.6.

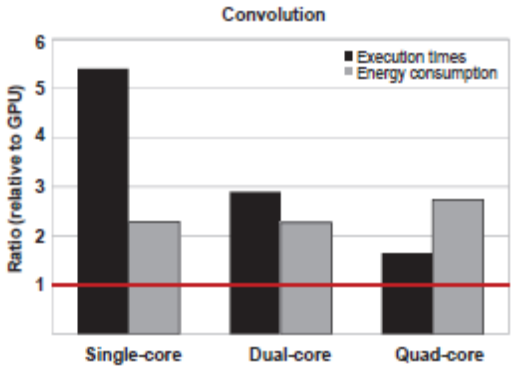
Dentre os testes realizados por Maghazed et. al. (2013), o mais interessante para os propósitos do presente trabalho trata do *benchmark* realizado para convolução, aonde os autores utilizaram como referência o algoritmo e o passo a passo fornecido por Gaster et. al. (2013), no livro *Heterogeneous Computing with OpenCL*. Maghazed et. al. (2013) fez a implementação do algoritmo para ser executado no SoC i.MX6 e na plataforma *Desktop*. Os resultados dos testes realizados são mostrados na Figura 22-a e na Figura 22-b. Na Figura 22-a são mostrados os tempos de execução, aonde o ganho de desempenho em código executado na GPU do SoC é de 5,45 vezes. Na Figura 22-b, são mostrados os ganhos de desempenho relativos à GPU, em comparação ao algoritmo executado em CPU com um núcleo, dois núcleos e quatro núcleos. A linha horizontal na Figura 22-b serve de base para o desempenho da GPU, sendo que qualquer

valor acima desta linha implica em menor tempo de execução e menor consumo de energia, comparativamente ao arranjo de código na CPU.

Convolution

	Time (ms)	Vivante GC2000	Nvidia Tesla M2050
Data Transfer	GPU kernel	1966	10.2
	CPU to GPU	143	13.5
	GPU to CPU	594	29
	Total (GPU)	2703	53
	Total (CPU)	14780	2166
	Speedup	5.45	40

(a)



(b)

Figura 22 - (a) Tempos de execução em ms para o algoritmo de Convolução. (b) Gráfico com desempenho relativo entre GPU e CPU para tempos de execução e consumo de energia. Linha vermelha corresponde à base de referência da GPU. Fonte: Maghazeh et. al (2013).

Ao final, Maghazeh et. al (2013) conclui que abordagens com implementação de uso da GPU resulta em soluções com maior eficiência energética, além de conseguir tempos médios de execução maiores do que abordagens em CPUs com 1 ou 2 núcleos de execução, porém, a GPU embarcada utilizada nos testes, a Vivante GC2000, possui pouca memória, suportando somente um arranjo de comandos de 512 instruções no total. Dessa forma, o maior limitante do uso de GPUs para processamento de dados em sistemas embarcados é a capacidade de memória da própria GPU para guardar instruções de execução. E depois, o segundo limitante é a quantidade e tamanho de registradores da GPU, pois reduzem a quantidade de *threads* ativas para execução de processos na GPU.

Um aspecto interessante do trabalho de Maghazed et. al. (2013) são as fórmulas fornecidas no trabalho para cálculo de consumo energético de uma configuração com GPU e com CPU e GPU, também chamada por configuração heterogênea. As fórmulas mencionadas são:

- $V_{GPU} * I^{GPU}_{benchmark} * T^{GPU}_{benchmark}$ = Potência consumida pela GPU durante os testes, aonde V é a tensão de alimentação do *chip* em Volts, $I^{GPU}_{benchmark}$ é a corrente consumida pela GPU durante o teste e $T^{GPU}_{benchmark}$ corresponde ao tempo gasto pela GPU para processamento das rotinas de teste, em segundos.
- $V_{GPU} * I^{GPU}_{benchmark} * T^{GPU}_{benchmark} + V_{CPU} * I^{CPU}_{benchmark} * T^{CPU}_{benchmark}$ = Potência consumida pelo arranjo CPU e GPU durante o processo de teste em um funcionamento heterogêneo, aonde trechos de código são executados tanto na CPU quanto na GPU, em que, além da fórmula anterior, temos em acréscimo um produto da tensão de alimentação da CPU (V_{CPU}) pela corrente consumida pela

CPU durante a execução do teste ($I_{benchmark}^{CPU}$) e pelo tempo gasto na execução de código na CPU ($T_{benchmark}^{CPU}$).

Em complemento aos trabalhos de Tristam e Bradshaw (2010) e Thouti e Sathe (2012), Ali (2013) realiza um estudo de modelos de programação paralela para computação *multicore*. Para isso, utilizou uma máquina com processador Intel Xeon E5345 com 8 núcleos a uma frequência de 2,33 GHz, em conjunto com uma GPU NVIDIA Tesla M2050.

Ali (2013) confirma que o desempenho de código paralelizado na GPU ultrapassa o desempenho obtido em código executado na CPU, e que para tarefas de mais baixo nível, OpenMP apresenta um desempenho melhor do que a biblioteca TBB, como mostrado com os dados de execução na Figura 23.

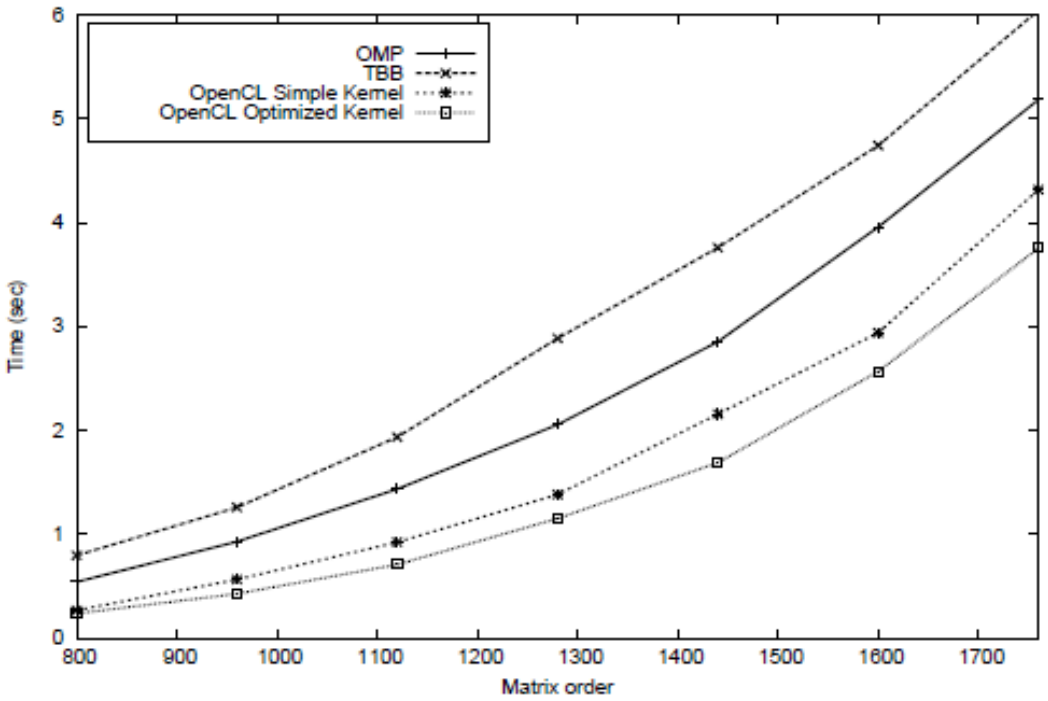


Figura 23 - Desempenho em multiplicação de matrizes de OpenMP, TBB e OpenCL. Fonte: Ali (2013).

Um detalhe importante a respeito do trabalho de Ali (2013) é que há a disponibilização dos códigos utilizados nas baterias de testes.

Apesar de a arquitetura dominante no mercado de dispositivos móveis ser a arquitetura ARM, a Intel está entrando neste mercado com sua plataforma Atom, que executa instruções x86 com um consumo compatível com os requisitos de sistemas móveis. Para fazer um comparativo entre processadores Intel Atom e ARM Cortex, Roberts-Huffman e Hedge (2009) rodaram uma bateria de testes em dois sistemas, uma BeagleBoard com um SoC OMAP3530, baseado no ARM

Cortex-A8, e outro em um computador Intel Atom N330. Um destaque para o SoC OMAP3530, é que este possui uma CPU ARM Cortex-A8, uma GPU SGX510 e um DSP C64x+, que pode ser programado e executar tarefas em conjunto com a CPU.

Roberts-Huffman e Hedge escolheram o sistema operacional Linux para ambos os sistemas, com Kernel Linux em versão 2.6.28. Na plataforma Intel, compilaram os programas de *benchmark* sem demais otimizações, mas para a plataforma ARM, compilaram os programas com as seguintes *flags*, que orientam o compilador sobre o uso de componentes internos:

—–mtune=cortex-a8 –mfpu=neon –ftree-vectorize – mfloat-abi=softfp

Sem as *flags* de compilação, os autores destacaram que o desempenho do *benchmark* Whetstone decai por um fator de duas vezes, e o desempenho do *benchmark* Linpak decai por um fator de 4.5. Já a plataforma Intel Atom N330 se saiu bem em todas operações, mesmo sem demais otimizações de compilação, porque todas as operações aritméticas usam a unidade de ponto flutuante (FPU) do processador.

Os testes escolhidos pelos autores foram os seguintes:

1. Fhourstone: Um *benchmark* de operações com números inteiros que efetivamente resolve o jogo "*Connect-4*" O *benchmark* calcula o número de posições pesquisadas por segundo.
2. Dhrystone: Outro *benchmark* de operações com números inteiros, projetado para testar uma gama de funções da máquina. Como saída, fornece o número de iterações *Dhrystone* executadas por segundo, também conhecidas por *Dhrystone* MIPS (DMIPS).
3. Whetstone: Um *benchmark* usado para medir desempenho computacional em cálculos científicos, fazendo emprego de operações com ponto flutuante. Como saída, fornece o número de Instruções *Whetstone* por segundo (WIPS).
4. Linpack: Um *benchmark* de operações em ponto flutuante, usado para medir o desempenho computacional com cálculos de álgebra linear, por meio do tempo gasto na resolução de equações lineares N X N. Como saída, esse *benchmark* fornece o número de operações em ponto flutuante por segundo (FLOPS).

Os resultados obtidos por Roberts-Huffman e Hedge (2009) são mostrados na Tabela 2, aonde pode ser observada a vantagem do Intel Atom N330 sob o ARM Cortex-A8 em todos os testes.

Tabela 2 - Tabela com resultados comparativos de benchmark entre a BeagleBoard e Intel Atom N330. Fonte: Roberts-Huffman e Hedge (2009)

	Fhourstone (Kpos/s)	Dhrystone* (DMIPS)	Whetstone (MIPS)	Linpack (KFLOPS)
Raw Benchmark Performance				
Beagle Board	731	883	100	23376
Atom 330 Desktop	2054	1822	1667	933638
Benchmark Performance per MHz (Beagle at 600MHz, Atom at 1.6GHz)				
Beagle Board	1.22	1.47	0.17	38.96
Atom 330 Desktop	1.28	1.14	1.04	583.52
Benchmark Performance per W (Beagle at 0.5W and Atom at 2W)				
Beagle Board	1462	1766	200	46752
Atom 330 Desktop	256.75	227.75	208.38	116704.75
Benchmark Performance per Dollar(Beagle at $32, Atom at $43)				
Beagle Board	22.84	27.59	3.13	730.5
Atom 330 Desktop	47.77	42.37	38.77	21712.51
Benchmark Performance per mm2 of the final die size (OMAP at 60mm2, Atom at 52mm2)				
Beagle Board	12.18	14.72	1.67	389.6
Atom 330 Desktop	39.5	35.03	32.06	17954.58

Porém, por se tratarem de arquiteturas diferentes, com frequências de operação diferentes, os autores optaram por analisar os resultados obtidos em comparação com a frequência de operação, com a potência consumida, custo do *chip* e tamanho do *chip*.

Quando avaliados sob a ótima de consumo energético, considerando que o ARM Cortex-A8 consome cerca de 0.5 W e o Intel Atom, 2 W, o processador ARM saiu em maior vantagem. Já com a avaliação dos resultados proporcionais ao preço do *chip*, o desempenho do processador Atom foi superior. Na época do trabalho publicado, o processador OMAP3530 custava 32 dólares, enquanto que o Atom N330 custava US$ 43 dólares, e sob essa ótica. O preço do OMAP 3530 é superior a demais SoCs disponíveis no mercado em decorrência do fato de possuir um DSP integrado. Por fim, avaliando os desempenhos obtidos em proporção com o tamanho do *chip*, o processador Atom N330 também se saiu à frente do ARM Cortex-A8.

Roberts-Huffman e Hedge (2009) concluem que a próxima geração de processadores ARM, a família Cortex-A9, deve obter desempenhos melhores e se aproximar mais ainda do desempenho da família Intel Atom. Além disso, destacam que como o foco da arquitetura ARM é o baixo consumo de energia, isso foi revelado nos testes, em que mesmo o ARM Cortex-A8 rodando a 600 MHz e consumindo 0.5 W, obteve um desempenho proporcional ao consumo de energia maior que o do Atom N330.

Com base no trabalho de Roberts-Huffman e Hedge, foram escolhidos os *benchmarks* de Dhrystone, Whetstone e Linpack para analisar e escalar as plataformas utilizadas no presente

trabalho, tendo por base uma análise comparativa do desempenho por frequência de operação e por custo do *chip*, e também por custo de plataforma, dada a disseminação de placas de desenvolvimento de tamanho custo reduzidos.

Considerando os avanços que uma CPU e um DSP integrados podem trazer, principalmente tendo em vista a busca por melhoria de desempenho em visão computacional, Coombs e Prabhu (2011) trabalharam numa modificação, feita pela empresa *Texas Instruments*, que adaptou a biblioteca OpenCV, de modo a permitir a execução de algumas de suas funções em DSPs da linha C6000, da mesma empresa. Coombs e Prabhu realizaram análises de desempenho para quantificar os ganhos obtidos com a biblioteca adaptada em um arranjo com processador ARM e DSP, contra um sistema contendo somente o processador ARM. Os autores destacam que um dos maiores problemas da utilização da biblioteca OpenCV em sistemas embarcados é a falta de suporte nativo à processamento de dados em ponto flutuante, o que é problema grave, considerando que a biblioteca faz uso de várias funções que dependem de cálculos em ponto flutuante.

Para demonstrar as diferenças de desempenho em decorrência da influência de suporte ou não à operações matemáticas com ponto flutuante, Coombs e Prabhu (2011) realizaram testes com imagens em resolução de 320x240 pixels, utilizando funções da biblioteca OpenCV que fazem uso de rotinas computacionais com ponto flutuante. Os testes foram realizados em um ARM9 do SoC OMAP-L138, um ARM Cortex-A8 do SoC DM3730, e no DSP C674x do SoC OMAP-L138, todos os núcleos foram colocados para executar em frequência de 300 MHz. Os resultados obtidos na bateria de testes foram agrupados e podem ser vistos na Tabela 3.

Tabela 3 - Avaliação de desempenho para algumas funções da biblioteca OpenCV com matemática de ponto flutuante. Fonte: Coombs e Prabhu (2011)

Function Name	ARM9™ (ms)	ARM Cortex-A8 (ms)	C674x DSP (ms)
cvCornerEigenValsandVecs	4746	2655	402
cvGoodFeaturestoTrack	2040	1234	268
cvWarpAffine	82	37	17
cvOpticalFlowPyrLK	9560	5240	344
cvMulSpecturm	104	69	11
cvHaarDetectObject	17500	8217	1180

O processador ARM9 não possui unidade para processamento em ponto flutuante, o que se revela ao fato que foi o processador que tomou mais tempo para completar as funções. Já o processador ARM Cortex-A8 possui coprocessador NEON, que corresponde à unidade de ponto flutuante, e por conta disso teve, em média, o dobro de desempenho do ARM9. Já o processador

DSP foi o que teve o melhor desempenho, já que sua arquitetura é otimizada para processamento pesado de dados matemáticos.

Outro fato a que os autores Coombs e Prabhu (2011) atentam, é ao fato de que em sistemas embarcados normalmente temos o arranjo de memória compartilhada, aos vários periféricos que integram um SoC, como por exemplo uma CPU e um DSP. Dessa forma, é preciso planejar bem o controle de acesso à memória, com destaque à largura de banda de acesso à memória, que é um dos fatores limitantes da velocidade de acesso.

Com base nisso, os autores decidiram investigar o comportamento da execução de três algoritmos de processamento de imagens, utilizando DMA para acessar memória externa, carregando elementos da imagem em sequência para a memória RAM, para serem processados, e utilizando também a forma de acesso convencional, feita sem DMA. Os resultados obtidos são mostrados na Tabela 4. Para a realização dos testes, foi utilizado o SoC OMAP3530 da *Texas Instruments*, operando a 720 MHz.

Tabela 4 - Tabela com demonstrativo de ganho de desempenho em processamento de imagens utilizando DMA e acesso direto. Fonte: Coombs e Prabhu (2011)

Function	Slice-based processing with DMA (ms)	In-place processing with cache (ms)
Gaussian filtering	6.1	7.7
Luma extraction	3.0	10.9
Canny edge detection	41.4	79.1

Como observado, o processamento realizado com uso de DMA para carregar os dados da imagem em memória RAM é mais eficiente que o acesso direto sem DMA

Coombs e Prabhu (2011) fazem a mesma afirmação que Pulli et. al. (2012), em que ainda há muito a que ser feito para que sistemas embarcados possuam desempenho em tempo real, em aplicações de visão computacional. Ambos os trabalhos ressaltam a necessidade de se utilizar, de modo mais eficiente, elementos aceleradores como unidades GPU e coprocessadores, assim como também DSPs, por exemplo.

No trabalho de Coombs e Prabhu (2011), os autores também descrevem, de maneira resumida, o trabalho tido pela adaptação de algumas rotinas da biblioteca OpenCV para execução nos processadores DSP da *Texas Instruments*. Para trabalhar nas chamadas para instruções do DSP, a equipe da *Texas Instruments* fez uso de uma suíte de desenvolvimento chamada C6EZAccel, que fornece uma API para realizar chamadas a instruções DSP por meio da camada de execução do processador ARM.

De modo a compreender a mudança provocada em um modelo arquitetural no qual um processador ARM processa uma aplicação em conjunto com um DSP, os autores fizeram a imagem mostrada na Figura 24. Nessa figura, à esquerda, é mostrada a sequência convencional de processamento paralelo em uma abordagem com uso somente de processador ARM. E na mesma imagem, à direita, é mostrada a abordagem em que trechos de processamento pesado da aplicação são transferidos para o processador DSP, ao mesmo tempo em que o processador ARM *multicore* pode executar também rotinas paralelas, trazendo ganho de desempenho e reduzindo tempo gasto na aplicação.

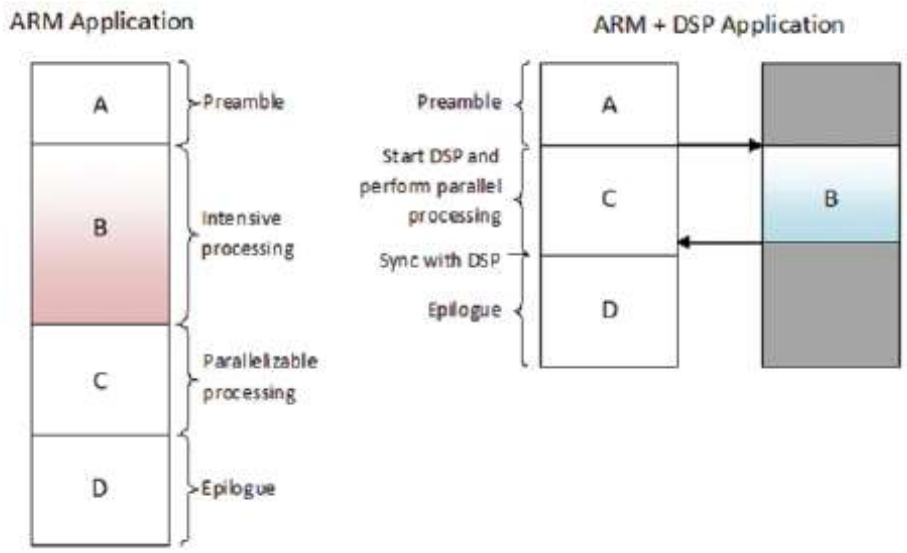


Figura 24 - Mudança arquitetural com funcionamento assíncrono entre ARM e DSP. Fonte: Coombs e Prabhu (2011).

Coombs e Prabhu (2011) destacam que fizeram pequenas modificações no código para interação com o processador DSP, e que há ainda muito trabalho a ser feito, principalmente em se tratando de modificações em baixo nível para melhorar acesso à memória e uso de recursos. Para demonstrar os ganhos obtidos com auxílio da biblioteca utilizada, os autores fizeram uma nova bateria de testes, executando uma sequência de funções da biblioteca OpenCV em uma imagem com resolução de 640 x 480 pixels, em processadores ARM Cortex-A8 e ARM Cortex-A8 com DSP c674x. Os processadores ARM rodaram os testes em frequência de 1 GHz, enquanto que o DSP operou à frequência de 800 MHz. Os resultados obtidos são mostrados na Tabela 5.

No mais, concluem que os resultados obtidos com pequenas otimizações já são bem promissores, sendo perceptível, em média, um aumento significativo de desempenho com relação à execução da biblioteca OpenCV em um arranjo lidando somente com o processador ARM.

Tabela 5 - Análise de desempenho de execução de bibliotecas OpenCV em ARM e ARM com DSP. Fonte: Coombs e Prabhu (2011)

OpenCV Function	ARM Cortex™-A8 with NEON (ms)	ARM Cortex-A8 with C674x DSP (ms)	Performance Improvement (cycle reduction)	Performance Improvement (x-factor)
cvWarpAffine	52.2	41.24	**21.0%**	**1.25**
cvAdaptiveThreshold	85.029	33.433	**60.68%**	**2.54**
cvDilate	3.354	1.340	**60.47%**	**2.50**
cvErode	3.283	2.211	**32.65%**	**1.48**
cvNormalize	52.258	14.216	**72.84%**	**3.68**
cvFilter2D	36.21	11.838	**67.3%**	**3.05**
cvDFT	594.532	95.539	**83.93%**	**6.22**
cvCvtColor	16.537	14.09	**14.79%**	**1.17**
cvMulSpectrum	89.425	15.509	**78.18%**	**5.76**
cvIntegral	8.325	5.789	**30.46%**	**1.44**
cvSmooth	122.57	57.435	**53.14%**	**2.14**
cvHoughLines2D	2405.844	684.367	**71.55%**	**3.52**
cvCornerHarris	666.928	168.57	**74.72%**	**3.91**
cvCornerEigenValsandVecs	3400.336	1418.108	**58.29%**	**2.40**
cvGoodFeaturesToTrack	19.378	4.945	**74.48%**	**4.29**
cvMatchTemplate	1571.531	212.745	**86.46%**	**7.43**
cvMatchshapes	7.549	3.136	**58.45%**	**2.48**

Infelizmente, um ponto negativo do arranjo ARM com DSP é o fator custo, que Coombs e Prabhu (2011) não destacaram. No trabalho, os autores fizeram uso do SoC TI TMSC6A816x C6-Integra[23], que possui processadores ARM Cortex-A8 e o DSP C674x integrados. Somente o SoC custava, na época do artigo, cerca de US$ 49,00. E um *kit*[24] de desenvolvimento com este *chip* custava 2494,99 dólares. Ou seja, são valores elevados, ainda considerando fatores como taxas de transporte e importação para o Brasil. A título de exemplo, a placa WandBoard Quad, usada no trabalho e que será melhor apresentada adiante, custa 129 dólares

3.2. Processadores ARM

O foco do presente trabalho é a arquitetura ARM, presente na grande maioria de dispositivos móveis e sistemas embarcados em geral (LANGBRIDGE, 2014). É uma arquitetura

[23] Maiores detalhes sobre o SoC, como *datasheets*, diagrama de blocos dos componentes, dentre outros, podem ser vistos neste link: http://focus-webapps.ti.com/docs/prod/folders/print/tms320c6a8167.html

[24] O *kit* mencionado é o DM816x/AM389x, que integra o SoC e demais componentes para desenvolvimento. Demais detalhes sobre o *kit* podem ser vistos neste link: http://www.ti.com/tool/tmdxevm8168

do tipo *Reduced Instruction Set Computer* (RISC), que envolve uma abordagem em que são implementados um menor número de instruções para o processador executar.

Essa abordagem resulta em custos menores de desenvolvimento, uma arquitetura mais simples, menor dissipação de energia e consumo elétrico, em suma, fatores favoráveis para sistemas embarcados.

A empresa ARM não fabrica seus processadores, porém, os licencia para que outras empresas possam modifica-los, personaliza-los e vendê-los conforme necessário, bastando adquirir os contratos de propriedade intelectual sobre os projetos de processadores (LANGBRIDGE, 2014).

Atualmente, os processadores ARM encontram-se nas versões ARMv7, ARMv8 e ARMv8-A, que são as arquiteturas 32-*bit* e 64-*bit* que agregam as tecnologias mais modernas, podendo envolver *chips* com mais de um núcleo, além de unidades de ponto flutuante VFP e NEON, que auxiliam em muito nas tarefas de processamento aritmético, multimídia e processamento digital de sinais (ARM, 2016).

3.3. Coprocessadores de ponto flutuante VFP e NEON

Inicialmente, processadores ARM da família ARM11 foram os primeiros a serem desenvolvidos por padrão com uma unidade de ponto flutuante, inicialmente chamada de *Vector Floating-point Architecture*, formalmente nomeada como VFPv2[25] pela ARM. Essa arquitetura implementa os padrões IEEE para processamento aritmético binário com ponto flutuante, estabelecidos pela norma IEEE 754. Em processadores ARM11 com unidade de ponto flutuante VFP, é necessário instruir o compilador com a *flag* "-mfpu=vfp", para utilizar a unidade.

A arquitetura NEON é uma tecnologia que permite o processamento de dados com o perfil *Single Instruction Multiple Data*, normalmente resumido pela sigla SIMD, que em termos de arquitetura de computadores quer dizer que uma mesma instrução, ou comando, pode ser aplicado a múltiplas entradas de dados. E a plataforma NEON também atua como unidade de ponto flutuante, de acordo com as especificações VFPv4 (LANGBRIDGE, 2014).

Basicamente, a arquitetura NEON compreende uma extensão do *set* de instruções ARM, possuindo uma base adicional de 32 registradores de 64 *bits*, ou, dependendo do arranjo, também podem ser configurados em 16 registradores de 128 *bits* (LANGBRIDGE, 2014).

[25] Maiores detalhes sobre as especificações VFPv2 desenvolvidas pela ARM podem ser encontradas aqui: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360e/BABHHCAE.html

Nesta arquitetura, os registradores são considerados vetores de elementos de um mesmo tipo de dado. E os dados permitidos são *signed/unsigned* 8-*bit*, 16-*bit*, 32-*bit*, 64-*bit* e 32-*bit float*, e as instruções desempenham a mesma operação em todas as trilhas concorrentes de registradores (LANGBRIDGE, 2014).

A título de exemplo, o esquemático arquitetural da tecnologia NEON é mostrado na Figura 25, que destaca os registradores origem, onde os dados são carregados na forma de elementos, e um mesmo conjunto de operações é aplicado sobre as trilhas sucessivas de registradores origem, e os resultados das operações são armazenados no registrador de destino.

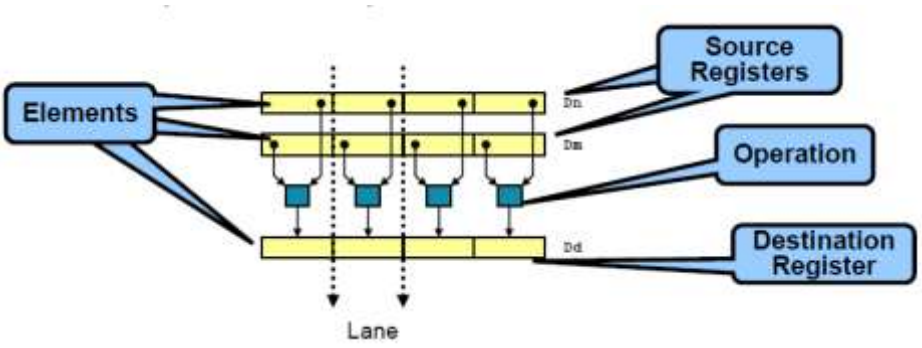


Figura 25 - Esquemático de funcionamento da arquitetura NEON. Fonte: ARM.com.

Os registradores utilizados pelo coprocessador NEON estão em um banco de memória de 256 bytes, separado do banco de registradores principal, normalmente denotado pelos registradores de r0 a r15. Além disso, é possível organizar o banco de registradores NEON em duas perspectivas diferentes, destacadas a seguir, e na Figura 26 é demonstrado o arranjo dos bancos em cada perspectiva.

- 32 registradores de 64 *bits*, correspondendo às unidades de D0 a D31;
- 16 registradores de 128 *bits*, correspondendo às unidades de Q0 a Q15.

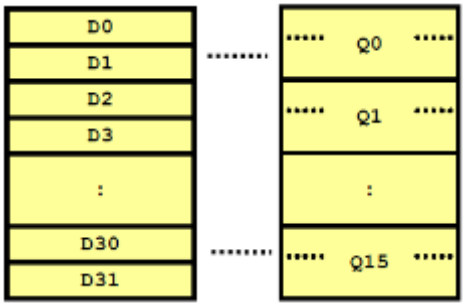


Figura 26 - Duas perspectivas diferentes de registradores da arquitetura NEON. Fonte: ARM.com.

Além disso, é possível fazer arranjos mistos de registradores, dependendo das operações a serem realizadas, permitindo que elementos de dados sejam eficientemente mantidos em precisões apropriadas para seus tipos. Ou seja, é possível realizar operações com dados de 32 *bits* e armazenar o resultado em registradores de 64 *bits*, ou o contrário também, realizar operações com dados de 64 *bits* e armazenar o resultado em registradores de 32 *bits*, como mostrado na Figura 27.

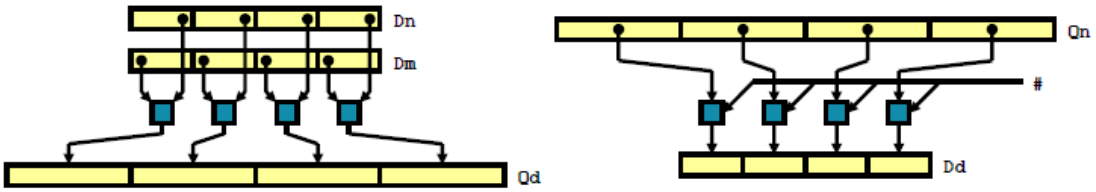


Figura 27 - Imagem ilustrativa de arranjos arquiteturais entre registradores com perspectivas diferentes no coprocessador NEON. Fonte: ARM.com.

As operações permitidas pelo coprocessador NEON são divididas nas seguintes categorias:

- Adição/Subtração
  - E operações relacionadas, como Mínimo, Máximo, Absoluto, Negação, etc.
- Multiplicação
- Comparação e Seleção
- Operações Lógicas (AND, OR, etc);
- Dentre outras.

Existem diversas formas de se programar e utilizar os recursos oferecidos pela plataforma, e consequentemente, coprocessador NEON. Em se tratando de um ambiente Linux Embarcado, em execução sob um SoC ARM compatível, temos as seguintes opções:

1) Utilizar diretivas de compilação com o compilador GCC
   As versões mais modernas do compilador GCC, como por exemplo, a versão 4.6.3, já possuem suporte para compilar rotinas para o coprocessador NEON, que normalmente é utilizado como unidade de ponto flutuante quando no uso de diretivas. Porém, utilizando diretivas de compilação, ou *flags* de compilação, todo o trabalho de otimização de código é deixado a cargo do compilador. Para

compilar um código utilizando os recursos do coprocessador NEON, basta utilizar os seguintes comandos, em sequência à chamada do compilador GCC:

- **-mfpu=neon** : Destaca que a unidade de ponto flutuante a ser utilizada é o NEON.
- **-ftree-vectorize** : Permite a *vetorização* automática de dados a serem armazenados nos registradores do coprocessador.

2) Usar rotinas *C Instrinsics*[26]

É possível também utilizar diretamente funções específicas em Linguagem C, que nada mais são do que um agrupamento de rotinas em *assembly* correspondentes às instruções NEON desejadas.

Para isso, é necessário especificar o uso da biblioteca NEON no código escrito em C, adicionando o comando "#include <neon.h>" no início do código.

3) Programar em Assembly[27]

Para casos em que se deseja o maior nível de otimização, é possível programar as rotinas diretamente em *assembly*, seja em código de programa puramente escrito em *assembly*, ou em código C com trechos em *assembly*.

As orientações fornecidas pela própria ARM, no que diz respeito ao uso de rotinas NEON em código C por meio de *C Intrinsics*, é utilizar a palava **__restrict** quando se tratar de ponteiros que devem ter espaços estritamente reservados em memória, e ao optar por usar as diretivas de compilação para vetorização automática de dados, faz *loops* múltiplos de 2 para auxiliar o processo automático.

3.4. Linux Embarcado

Linux Embarcado tipicamente se refere a um sistema completo, no contexto de uma distribuição ajustada para dispositivos embarcados. Em termos gerais de Kernel e de aplicações, praticamente não há diferença entre os códigos utilizados em um ambiente *desktop* do código

[26] É possível encontrar mais detalhes sobre o uso das rotinas *C Intrinsics* da arquitetura NEON no centro de informações da ARM, disponível em:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0205g/Ciabijcc.html

[27] Todo o conjunto de instruções *assembly* suportadas pela arquitetura NEON está disponível em:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489c/CJAJIIGG.html

utilizado em um ambiente embarcado. Basicamente, são as mesmas rotinas compiladas para plataformas específicas, contando com pacotes e *drivers* necessários para uma dada aplicação (YAGHMOUR et. al., 2008).

A diferença, contudo, aparece no contexto de aplicações de tempo-real, em que as tarefas realizadas pelo sistema operacional possuem janelas bem específicas de tempo para sua realização, característica essa muito necessária em diversos cenários de aplicações de sistemas embarcados. Para atender esse contexto, o *kernel* do sistema Linux precisa ser adaptado por meio de um *patch* chamado **PREEMPT_RT** (Linux.com, 2016).

Um sistema *Linux* embarcado é composto basicamente por três componentes principais: *Bootloader*, *Kernel* e *root filesystem*. Basicamente, o *bootloader*, localizado na primeira região de memória, é responsável por preparar o *hardware* para a execução do *Kernel*. O *Kernel* é o que gerencia os dispositivos de *hardware* e a interação desses com os *softwares* em execução. Já os *softwares* e demais bibliotecas, arquivos e configurações de sistema e de usuário, ficam armazenados no *root filesystem*, ou sistema de arquivos raiz, em português, que basicamente é o espaço reservado para armazenamento permanente de arquivos no sistema. A Figura 28 mostra o arranjo e a ordem básica desses componentes na memória de um sistema *Linux* embarcado (YAGHMOUR et. al., 2008).

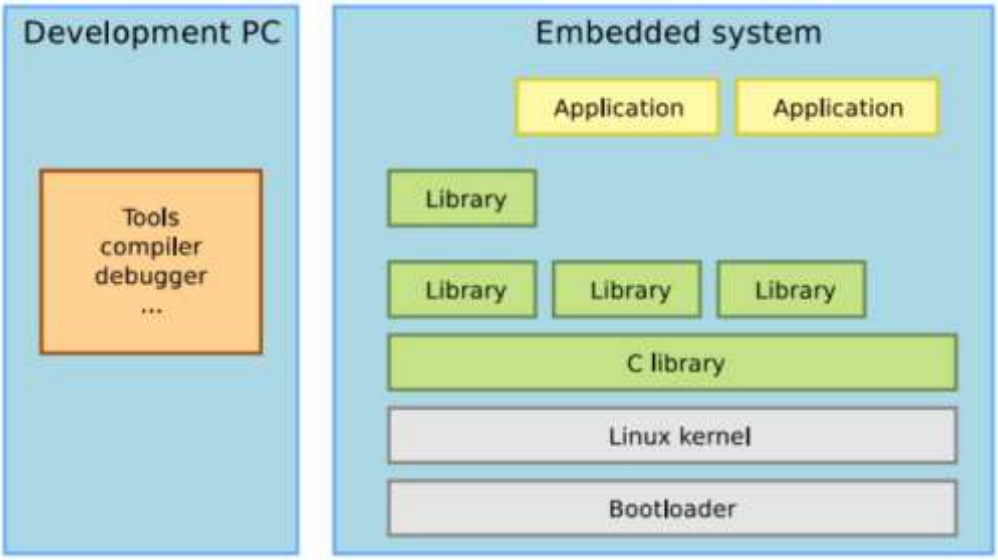


Figura 28 - Esquemático de diferenças entre um computador de desenvolvimento e um sistema embarcado com Linux. Fonte: free-electrons.com

Para desenvolver aplicações e/ou sistemas completos para ambientes Linux embarcados em plataformas ARM, a principal metodologia é o uso de *toolchain* para compilação cruzada, do inglês, *cross-compilation*, que significa compilar uma aplicação para uma arquitetura diferente da máquina utilizada, como mostrado na Figura 29.

Toolchain[28] nada mais é do que um conjunto de ferramentas para compilação e depuração de código. Cada *toolchain* fornece basicamente 5 elementos:

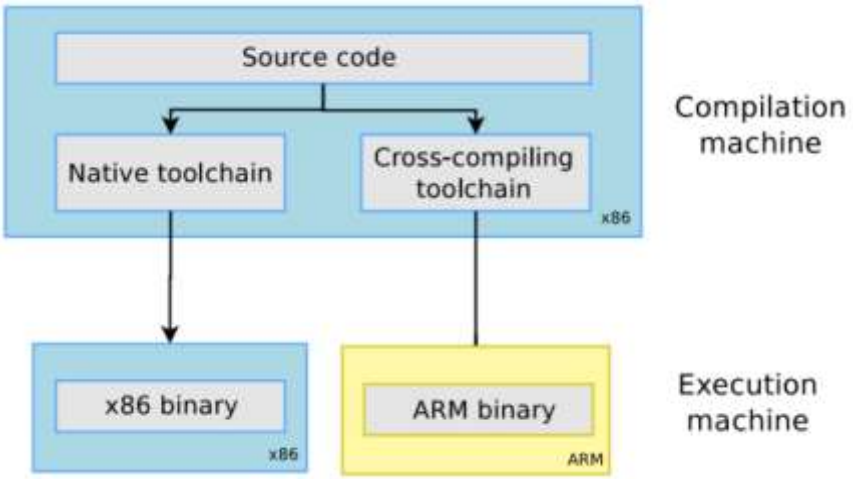


Figura 29 - Esquemático de diferença entre compilação nativa e compilação cruzada. Fonte: free-electrons.com.

1. **Compilador GCC:** É o *GNU C Compiler*, ou Compilador C GNU, traduzido para o português. É compatível com diversas linguagens, como C++ e Java, além de conseguir gerar código para arquiteturas como ARM, x86, MIPS, PowerPC, dentre outras.
2. **Binutils:** É um conjunto de ferramentas para manipular programas binários[29] para arquiteturas específicas, como a ferramenta "as" para ler código-fonte em linguagem *assembly* e gerar um arquivo objeto[30] correspondente, e a ferramenta "ld", responsável por fazer a ligação entre um ou mais arquivos objetos em um arquivo executável.
3. **Biblioteca C Padrão:** É a biblioteca responsável por fazer a interface do programa com o *Kernel Linux* através das chamadas do sistema (*System Calls*). Essa biblioteca possui duas implementações mais famosas: a "glibc", mais usadas em ambientes *Desktop* por ser otimizada para *performance*, e a "uClibc", mais usada em dispositivos embarcados, por ser otimizada para gerar código menor.
4. ***Kernel Headers*:** Pelo fato de a biblioteca C padrão conversar com o *kernel* através de chamadas de sistema, é preciso saber quais são as chamadas a serem utilizadas. Elas estão descritas nos *Kernel Headers*, ou cabeçalhos do *Kernel*, de cada versão disponível. Além disso, a biblioteca C padrão precisa ter acesso às constantes e estruturas de dados do *Kernel*.

[28] *Toolchain*, traduzido para o português, cadeia de ferramentas. Quando se trata de programação, faz menção ao conjunto de utilitários para compilar, linkar e criar binários executáveis para arquiteturas diversas.
[29] Binários são tipos de arquivos codificados em forma binária, que correspondem aos zeos (0) e uns (1).
[30] Arquivo objeto é um tipo de arquivo que contém código de máquina realocável, que normalmente não é um executável, mas serve de base para compor arquivos executáveis.

5. **GDB:** O GDB é o *debugger* padrão utilizado na plataforma *Linux*. Ele não é necessário para compilar aplicações, mas ajuda muito a encontrar problemas em aplicações compiladas. Também auxilia na melhoria e na otimização de aplicações, pois é capaz de mostrar uso de memória e de chamadas ao sistema, o que serve de base para programadores melhorarem suas aplicações.

Obs.1: Uma observação, que serve tanto para a Biblioteca C padrão quanto que para as demais bibliotecas utilizadas, é que as bibliotecas do dispositivo precisam ser as mesmas do *toolchain* utilizado para gerar o sistema ou as aplicações para o sistema, do contrário, ocorrerão erros na execução das aplicações. Além disso, é preciso também contar com os mesmos *Kernel Headers* da máquina do dispositivo. Isso é necessário porque a comunicação da aplicação com o ambiente de sistema depende da biblioteca C e dos *Kernel Headers*, como mostrado na Figura 30.

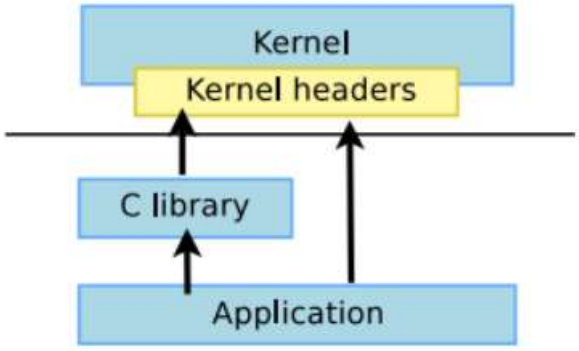


Figura 30 - Esquemático de interação entre os componentes de um sistema Linux. Fonte: free-electrons.com.

Obs.2: Existe o tipo de compilação que faz uso de *linkagem*[31] estática e *linkagem* dinâmica. A *linkagem* estática é aquela que agrega todas as bibliotecas utilizadas pela aplicação no binário executável gerado. Nesse caso, o tamanho do arquivo binário gerado é maior, mas a chance de ocorrer problemas com bibliotecas incompatíveis é inexistente. Com a *linkagem* dinâmica, o compilador gera o binário executável com referência a locais específicos no sistema, onde Estarão as bibliotecas a serem utilizadas, que são carregadas em tempo de execução quando a aplicação é iniciada. Essa forma, porém, é suscetível a casos em que o computador ou dispositivo pode não possuir as bibliotecas necessárias ou equivalentes, o que resulta em erro de execução.

Para interagir entre uma máquina base, normalmente chamada na literatura por *host*, e a máquina alvo, chamada na literatura por *target*, que é a máquina que possui o sistema Linux embarcado e adaptado para sua arquitetura, normalmente usa-se comunicação Serial ou via rede,

[31] *Linkagem* é um estrangeirismo adaptado ao português, que vem do termo em inglês *linkage*, que significa unir coisas, ou a ação de unir coisas.

por meio de comunicação Ethernet. Na Figura 31 é mostrado um esquemático visual destes tipos de arranjos de comunicação.

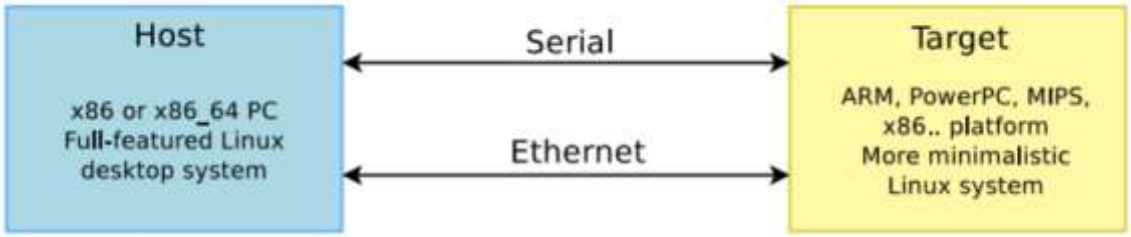


Figura 31 - Esquemático de interação entre a máquina base e a máquina alvo. Fonte: free-electrons.com.

Uma ferramenta muito bem-vinda para o desenvolvimento de aplicações com Linux Embarcado é o Valgrind[32], pois permite analisar o correto uso de memória, comunicação e execução de tarefas, dentre outras características, pela aplicação embarcada. Ou seja, usando o Valgrind é possível descobrir se a aplicação terá algum problema quanto ao uso de ponteiros e índices de memória, por exemplo.

E ainda no contexto de Linux Embarcado, outra ferramenta que também merece destaque é o Gprof[33], pois permite analisar com detalhes toda a estrutura de execução da aplicação embarcada, informando os tempos de execução tomados por cada parte do programa.

3.5. OpenCV

OpenCV[34] é uma biblioteca de processamento de imagens digitais e de visão computacional, de código aberto, amplamente utilizada pelo fato de possuir uma grande gama de funções implementadas, que facilitam a criação de rotinas e algoritmos de visão computacional.

Para que se tenha uma ideia, a biblioteca OpenCV possui mais de 2500 algoritmos de visão computacional otimizados, prontos para uso, tais como reconhecimento de face, inversões de cores, aplicações de filtros, etc.

Com algumas poucas linhas, já é possível ter uma aplicação capaz de obter uma imagem de um dispositivo de captura de imagem, como uma câmera USB, e realizar algum tipo de processamento nessa imagem.

[32] Página da ferramenta: http://valgrind.org/
[33] Página da ferramenta: https://sourceware.org/binutils/docs/gprof/
[34] Página do projeto: http://opencv.org/

É compatível com linguagens C e C++, Java e Python. O uso de OpenCV em C ou C++ é, porém, mais recomendável, por possuir maior desempenho e permitir a realização de operações matemáticas pesadas com paralelismo quando executado em computadores com mais de um núcleo.

A biblioteca OpenCV possui uma estrutura modular, que significa que a biblioteca possui diversas bibliotecas compartilhadas. Os seguintes módulos estão disponíveis na biblioteca:

- **core** – Um modulo compacto, com definições de estruturas de dados básicas, como matrizes multidimensionais e algumas outras funções básicas utilizadas pelos outros módulos.
- **imgproc** – É o modulo de processamento de imagens que inclui filtros de imagens lineares e não lineares, transformações geométricas de imagens, conversão de espaços de cores, histogramas, dentre outros.
- **video** – É o modulo que inclui algoritmos para análise de vídeo, com rotinas para estimação de movimento, remoção de fundo, e também rotinas para acompanhamento de objetos.
- **calib3d** – Módulo com algoritmos para visualização de imagens 3D ou com perspectiva estéreo, com rotinas para estimação de posição de objetos e reconstrução 3D.
- **features2d** – Módulo com rotinas para detecção e extração de características e descritores de imagens.
- **objdetect** – Módulo com rotinas para detecção de objetos e instâncias de classes predefinidas para trabalhar com detecção de face, olhos, pessoas e carros, por exemplo.
- **highgui** – Corresponde à interface para captura de vídeo e imagem, além de *codecs* associados, como também rotinas para criação de simples interfaces gráficas de usuário.
- **gpu** – Algoritmos para aceleração de algoritmos com execução na GPU.

A biblioteca OpenCV já em sua versão 2.4 possui compatibilidade com OpenGL, OpenCL e CUDA. Todavia, essa compatibilidade só está disponível em ambientes Windows e Linux desktop, ou seja, ainda não está implementada para uso em sistemas embarcados num contexto de Linux Embarcado para processadores ARM. Dessa forma, para usar OpenGL com OpenCV em Linux Embarcado, torna-se necessário a implementação e execução manual das rotinas necessárias. (OpenCV, 2016).

3.6. OpenMP

OpenMP[35] surgiu como uma plataforma facilitadora para o desenvolvimento, projeto e programação de aplicações paralelas em ambiente de memória compartilhada, fornecendo uma interface de programação de aplicação (do inglês, API) que consiste em:

- Diretivas de compilação
- Rotinas em tempo de execução
- Variáveis de ambiente

Em suma, os desenvolvedores do projeto queriam fornecer uma plataforma que permitisse ao programador desenvolver aplicações paralelas sem precisar saber como criar, sincronizar, destruir *threads*, e nem mesmo precisar determinar quantas *threads* de aplicação utilizar. Para alcançar isso, OpenMP faz uso de uma série de *pragmas*, diretivas, chamadas de função e variáveis de ambiente, fornecendo instruções ao compilador sobre como e onde inserir *threads* na aplicação.

Por exemplo, para paralelizar um laço de repetição, basta orientar o compilador com a diretiva "#pragma omp parallel for", acima de um laço do tipo *for*, como mostrado no trecho de código mostrado a seguir, onde é feita a conversão paralela de *pixels* em 32-*bit* RGB em *pixels* 8-*bit* em escala de cinza.

```
#pragma omp parallel for
for (i=0; i < numPixels; i++)
{
  pGrayScaleBitmap[i] = (unsigned BYTE)  (pRGBBitmap[i].red * 0.299 +
          pRGBBitmap[i].green * 0.587 +   pRGBBitmap[i].blue * 0.114);
}
```

Sem a diretiva, o laço mostrado anteriormente seria executado de maneira sequencial. E de modo a ilustrar a diferença provocada pela execução sequencial, da execução paralela, é mostrada a Figura 32, que ilustra a diferença de funcionamento em ambos os casos, para o laço do tipo *for*.

[35] Mais detalhes, instruções e até mesmo tutoriais de uso encontram-se na página *web* do projeto, disponível em: http://openmp.org/

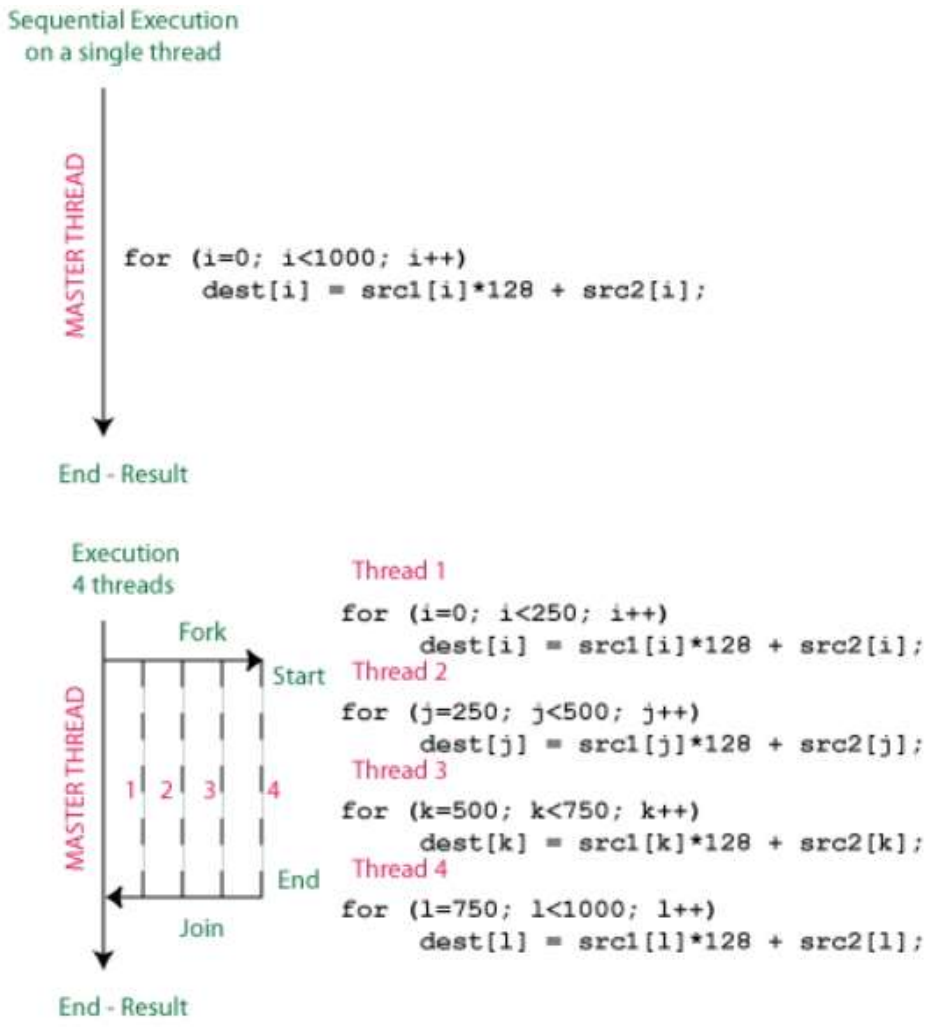


Figura 32 - Estrutura simplificada de funcionamento da biblioteca OpenMP. Fonte: CodeProject.com.

Para se instalar a biblioteca OpenMP em um ambiente Linux compatível com a distribuição *Debian*, pode-se proceder com o seguinte comando:

```
$ sudo apt-get install libgomp1
```

O pacote *libgomp1* corresponde à implementação GNU das especificações OpenMP. E além disso, para utilizar dos recursos de OpenMP no código, é necessário incluir a biblioteca no código escrito em linguagem C ou C++, com a seguinte linha no início do código:

```
#include <omp.h>
```

3.7. Threading Building Blocks

A biblioteca *Intel Threading Building Blocks*[36] é resultado de uma iniciativa da Intel de prover recursos de paralelização de código com um perfil de abstração em nível de orientação a objetos, para o programador.

Dessa forma, a biblioteca somente pode ser utilizada em código programado na linguagem C++, e em vez de o programador criar rotinas fazendo uso nativo de *threads*, a biblioteca abstrai o acesso a múltiplos processadores permitindo o acesso às operações por meio de tarefas, comumente chamadas na literatura específica de *tasks*, que são alocadas dinamicamente aos núcleos do processador, em tempo de execução, pela biblioteca.

Em suma, Threading Building Blocks assemelha-se bastante à abordagem da biblioteca OpenMP, pois também faz uso de pragmas e diretivas de compilação, ficando a maior diferença entre as duas bibliotecas no uso de *tasks* como mecanismo de abstração de recursos de paralelismo, como utilizado pela primeira.

3.8. OpenCL

OpenCL é a abreviação de *Open Computing Language*, e é um *framework* de programação heterogênea desenvolvido pelo consórcio tecnológico sem fins lucrativos Khronos. OpenCL é uma plataforma que visa permitir o desenvolvimento de aplicações que possam ser executadas em uma ampla gama de dispositivos, de diferentes fabricantes (Gaster et. al. 2013).

A especificação OpenCL suporta diversos níveis de paralelismo e mapeamento de sistemas homogêneos ou heterogêneos, consistindo de arranjos de CPUs e GPUs, e a possibilidade de executar códigos de programas em ambos os arranjos, como ilustrado na Figura 33.

E, recentemente, fabricantes de SoCs passaram a implementar processadores compatíveis com as especificações OpenCL, o que permite a criação de programas capazes de executar códigos em CPU e em GPU, simultaneamente, resultando em ganho de desempenho.

[36] *Threading Building Blocks* é uma biblioteca para paralelização de código C++ desenvolvida e mantida pela Intel. Mais detalhes em: https://www.threadingbuildingblocks.org/

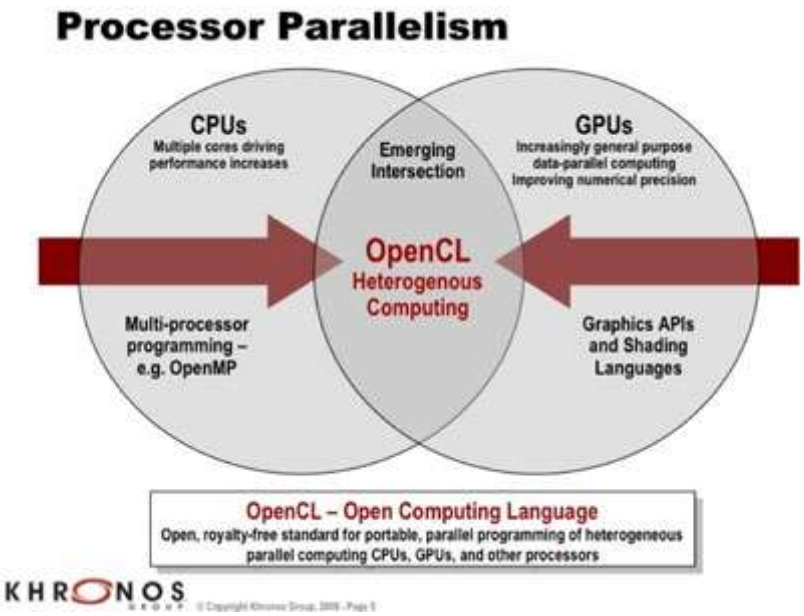


Figura 33 - Desmontração visual dos objetivos do framework OpenCL. Fonte: Khronos.com.

A estrutura do *framework* OpenCL determina uma entidade hospedeiro, chamada de *host*, e demais entidades dispositivos, chamadas de *devices*. Cada entidade dispositivo é caracterizada por um grupo de unidades computacionais, que podem ser CPUs ou GPUs. Em suma, a unidade hospedeiro é quem configura, coordena e comanda o funcionando das entidades dispositivo, como pode ser observado na Figura 34.

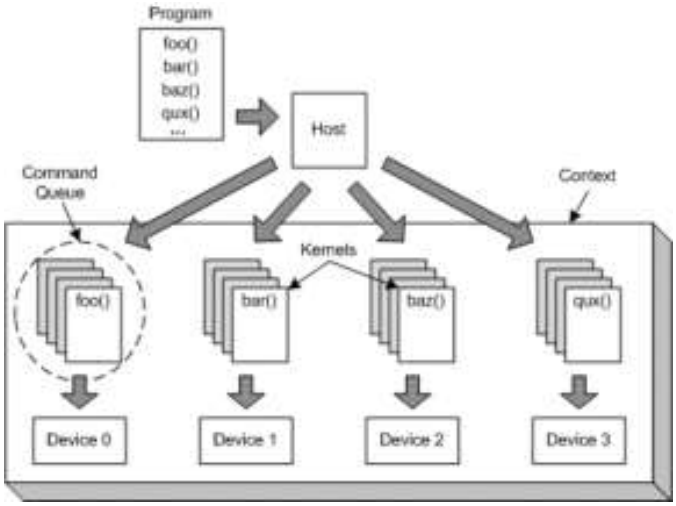


Figura 34 - O modelo da plataforma define uma arquitetura abstrata para os dispositivos. Fonte: www.drdobbs.com.

A máquina hospedeiro pode ser a própria máquina que possui as entidades dispositivos, como por exemplo, um computador *desktop* dotado de uma CPU e uma GPU compatíveis com OpenCL. E o código a ser executado em cada entidade dispositivo é chamado de *kernel*, que

basicamente engloba o conjunto de instruções a serem executadas em cada entidade, como também mostrado na Figura 34.

Um ponto importante a ser destacado é que o dispositivo precisa ser compatível com as especificações OpenCL, para permitir a execução de código coordenado por meio deste *framework*. E além da compatibilidade, o sistema operacional da máquina que estiver coordenando as entidades precisa possuir os *drivers* dos dispositivos, caso contrário, não será possível executar códigos OpenCL em uma GPU, por exemplo, por falta dos *drivers*.

3.9. OpenGL ES 2.0

O OpenGL ES envolve um conjunto de rotinas para computação gráfica voltada para sistemas embarcados, baseada na OpenGL originalmente desenvolvida para ambientes *Desktop* mas com diferenças no tratamento de chamadas a unidade de processamento gráfico - GPU. Sua versão 2.0 trouxe uma maior flexibilidade no desenvolvimento de aplicações e jogos por meio da criação de programas chamados Vertex Shaders e Fragment Shaders, que permitem instruir a GPU quanto à processos a serem executados sobre os gráficos trabalhados (KHRONOS, 2016). A programação dos chamados *Shaders* já era possível em ambiente Desktop, mas só foi possível em dispositivos móveis a partir da versão 2.0 da API embarcada do OpenGL, que é a OpenGL ES 2.0.

Veja na Figura 35 um esquemático de relação entre os elementos envolvidos, de acordo com Kessenich et. Al. (2012) e Freescale (2015):

- **Vertex Shader:** é o programa que deve ser carregado para computar operações sobre cada vértice do gráfico:
    - **Attributes**: Consiste nos dados de cada vertex usando arrays.
    - **Uniforms**: Dados que serão constantemente usados pelo programa vertex shader.
    - **Shader program**: É o código-fonte do Vertex Shader. Contém as instruções de operação que são executadas pelo Vertex.

- **Fragment Shader:** é o programa que deve ser carregado para computar operações sobre texturas (ou pixels) no gráfico:
    - **Varying variables:** São as saídas geradas pelo programa vertex shader, que são geradas pela unidade de rasterização para cada fragmento usando interpolação.

- **Uniforms**: Dados constantes que serão usados pelo fragment shader.
- **Samplers**: É um tipo específico de Uniform que representa a textura usada pelo fragment shader.
- **Shader program:** É o código-fonte do Fragment Shader que determina as operações que serão realizadas pelo Fragment.

Sendo assim, com base nas descrições de vertex e fragment shader, e com base no esquemático da Figura 35, pode-se relacionar as estruturas da seguinte forma:

- **Vertex Shader:** Toma por base variáveis do tipo *attribute*, que são computadas a cada vértice, e *uniform*, que são referências gerais, e gera como saída **gl_Position**, que é usado para definir, dados os processos realizados pelo shader, a posição do vértice computado.
- **Fragment Shader:** Recebe as variáveis do tipo *varying* do Vexter Shader, além de trabalhar com variáveis do tipo *uniform* para lidar com informações de textura, dentre outras propriedades, e por fim gera como resultado **gl_FragColor**, que serve de referência para renderizar a cor do fragmento processado.

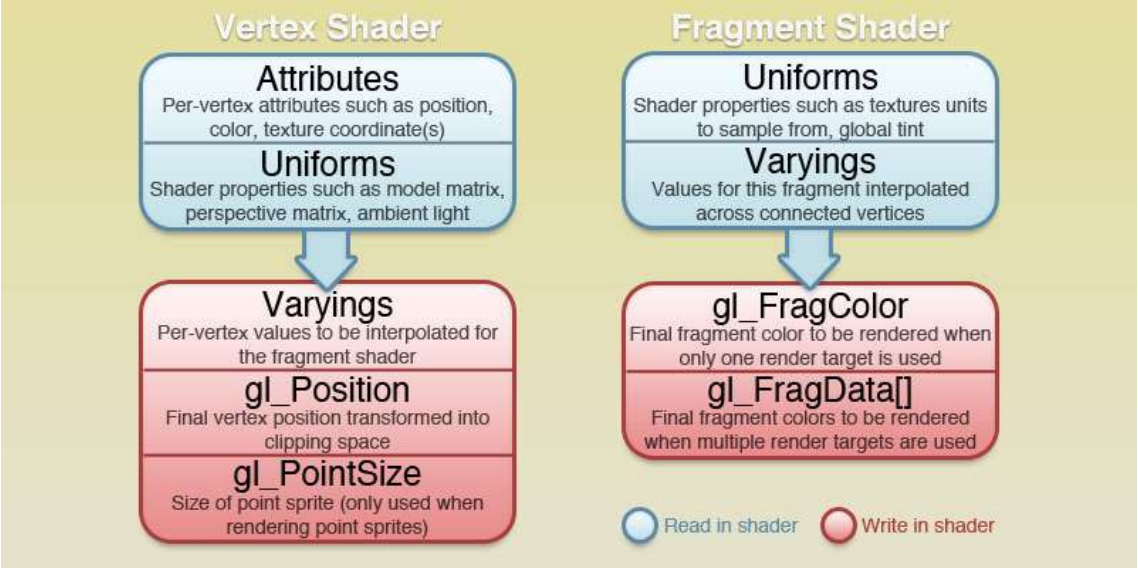


Figura 35 - Elementos chaves do OpenGL - Vertex Shader e Fragment Shader. Fonte: http://glslstudio.com.

De forma a ilustrar o processo de operação da execução de rotinas com OpenGL ES 2.0, veja na Figura 36 a sequência de operações realizadas. Nesta sequência os pontos mais importantes são o Vextex Shader, onde há aplicação das rotinas programadas em cada vértice do

gráfico, e o Fragment Shader, onde há aplicação das rotinas programadas em cada fragmento. Ambos os pontos estão destacados na cor laranja, na Figura 36.

Por fim do processo, o resultado é levado ao FrameBuffer, elemento-chave do sistema com GPU para exibição dos gráficos em vídeo (FREESCALE, 2015 e KESSENICH et. Al. 2012).

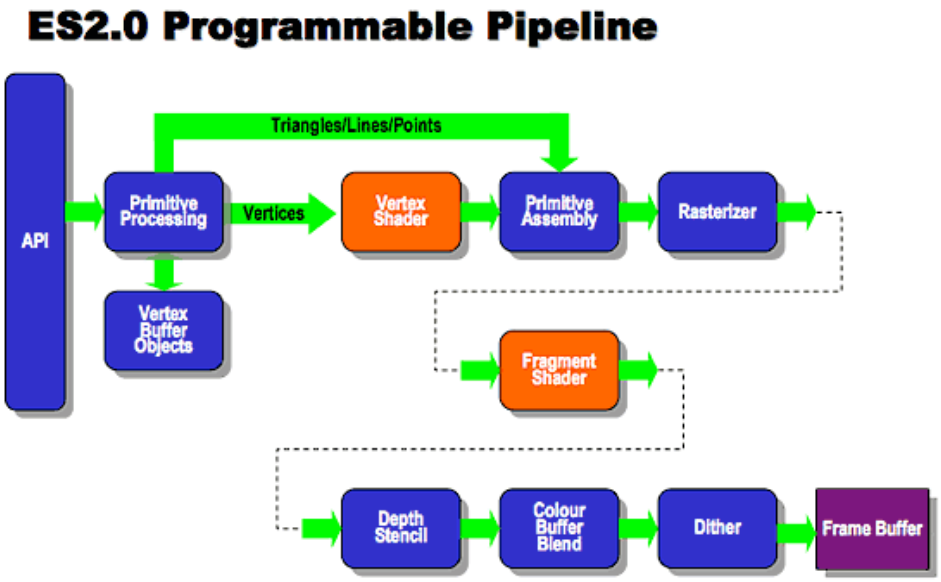


Figura 36 - Pipeline de operação do OpenGL ES 2.0. Fonte: https://www.khronos.org/.

Por realizar operações ponto-a-ponto, o Fragment Shader torna-se uma peça fundamental para a realização de Processamento de Imagem em GPU (FREESCALE, 2015), pois trabalha diretamente com as propriedades da imagem.

Observando com mais detalhes o processo de funcionamento entre o processamento de vértices e fragmentos, temos como ilustração a Figura 37. Nesta figura, os vértices (Vertices) de uma imagem/cenário qualquer passam por um processo onde são trabalhados com base em propriedades específicas (o que é feito pelo *Vertex Shader*), o que gera por resultado a etapa seguinte com os vértices transformados (Trans. Vertices). Estes vértices já possuem propriedades que permitem fazer as suas conexões/ligações, o que é feito na etapa de montagem (Assembly), e logo em seguida as conexões estão prontas para serem transformadas em regiões, o que é feito pelo processo de *raster*, e resulta nos fragmentos (Fragments). Na etapa dos fragmentos é possível trabalhar nas regiões definidas entre os limites dos vértices, com acesso às propriedades de cores sobre os elementos. É nesse ponto que entra o *Fragment Shader* para poder realizar aplicações de filtros, transformações de cores, dentre outros efeitos diversos, de interesse à área de Processamento de Imagens. Os resultados gerados pela etapa de fragmentos são passados por um processo de interpolação, que resultam então nos fragmentos da imagem.

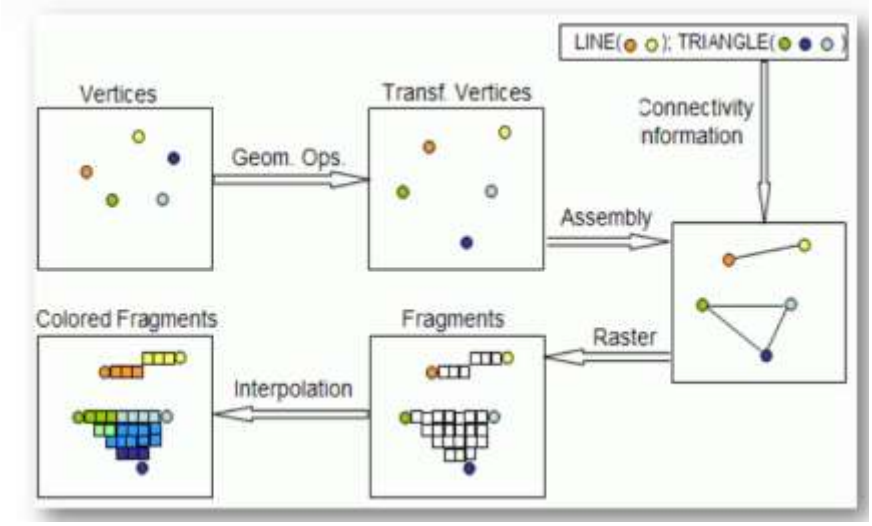


Figura 37 - Fluxo de Operação para Gráficos OpenGL ES 2.0. Fonte: www.nxp.com.

Em termos de processo funcional, o fluxo é tal como apresentado. Mas em termos de aplicação com código-fonte, é preciso seguir uma sequência aos moldes da sequência apresentada na Figura 38.

Nesta sequência, o fluxo normal de operação de programas com OpenGL e Shaders de programação é o seguinte:

1. Chamada da rotina **glCreateProgram()** – Cria uma referência de memória para criação e associação de programas Shaders a serem inseridos na GPU.
2. Chamada de **glAttachShader()** – Essa rotina deve ser chamada para vincular (ou anexar) tanto **Vertex Shader** quanto **Fragment Shader**. Na Figura 38 ela é chamada para associar cada Shader.
    a. Antes de realizar o **glAttachShader()**, cada Shader deve passar por um processo para ser criado, o que é feito pelo **glCreateShader()**, associado ao código-fonte do Shader em questão, o que é feito pelo **glShaderSource()**, e compilado para referência, o que é feito pelo **glCompileShader()**.
3. Anexados os Shaders ao programa OpenGL que será executado na GPU, é necessário chamar a rotina **glLinkProgram()** para vincular os objetos compilados ao programa OpenGL que será executado na GPU.
4. Por fim, a chamada do **glUseProgram()** é a responsável por carregar os códigos de Vertex e Fragment Shader na GPU para execução das rotinas OpenGL. Neste ponto, os códigos estão prontos para execução nos pontos destacados nas Figura 36 e 37.

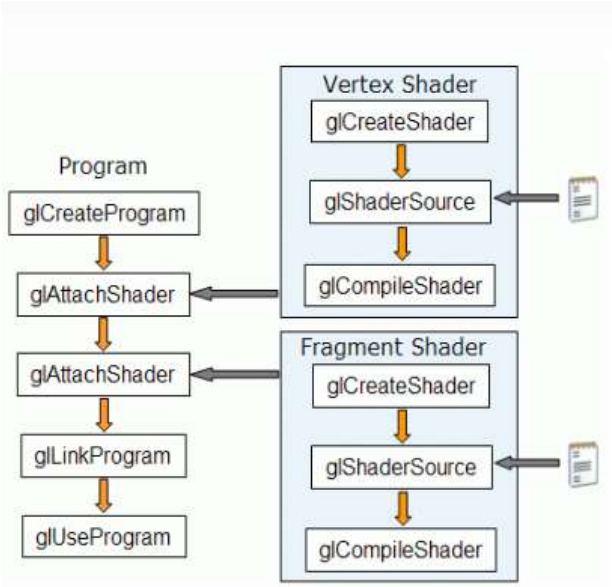


Figura 38 - Estrutura de Shaders para OpenGL ES 2.0. Fonte: www.nxp.com

A despeito de ter apresentado a tecnologia de OpenCL EP, que também usa recursos de GPU em alguns dispositivos, o OpenGL ES 2.0 possui maior abrangência, isto é, há mais dispositivos embarcados compatíveis com OpenGL ES 2.0 do que com OpenCL. Veja por exemplo na Tabela 6 uma listagem de dispositivos com GPUs compatíveis com OpenGL ES 2.0/3.0 disponíveis no mercado, tanto em se tratando de dispositivos Android, quanto iOS e até mesmo a Raspberry Pi.

Tabela 6 - Dispositivos com GPU compatível com OpenGL ES 2.0/3.0 – Análise em 2013 . Fonte: http://www.geeks3d.com

Device name	OS Version	SoC	GPU Vendor	GPU Renderer
Samsung Galaxy S	Android 2.3.3	Samsung Exynos 3 Single (3110)	Imagination	PowerVR SGX540
Samsung Galaxy S II	Android 4.1.2	Samsung Exynos 4 Dual (4210)	ARM	Mali-400 MP4
HTC Sensation	Android 4.0.3	Qualcomm Snapdragon S3 (MSM8260)	Qualcomm	Adreno 220
LG Optimus 2X	Android 2.3.4	Nvidia Tegra 250 AP20H	Nvidia	Tegra 2 AP20H
Asus Transformer Prime	Android 4.1.1	Nvidia Tegra 3 T30	Nvidia	Tegra 3 T30
Samsung Galaxy Nexus	Android 4.3	Texas Instruments OMAP4460	Imagination	PowerVR SGX540
Motorola RAZR I	Android 4.0.4	Intel Atom Z2480 Medfield	Imagination	PowerVR SGX540
Asus Nexus 7	Android 4.2.1	Nvidia Tegra 3 T30L	Nvidia	Tegra 3 T30L
HTC Desire X	Android 4.1.1	Qualcomm Snapdragon S4 Play (MSM8225)	Qualcomm	Adreno 203
Samsung Galaxy S II Plus	Android 4.1.2	Broadcom BCM28155	Broadcom	VideoCore IV
Raspberry Pi	Debian 7	Broadcom BCM2835	Broadcom	VideoCore IV
LG Nexus 4	Android 4.3	Qualcomm Snapdragon S4 Pro (APQ8064)	Qualcomm	Adreno 320
Samsung Nexus 10	Android 4.3	Samsung Exynos 5 Dual (5250)	ARM	Mali-T604
Samsung Galaxy S III	Android 4.1.2	Samsung Exynos 4 Quad (4412)	ARM	Mali-400 MP4
Apple iPad 3rd gen	iOS 6.1.3	Apple A5X	Imagination	PowerVR SGX543MP4
Apple iPhone 5	iOS 6.1.4	Apple A6	Imagination	PowerVR SGX543MP3
Nvidia Shield	Android 4.2.1	Nvidia Tegra 4	Nvidia	Tegra 4

Optou-se por trabalhar com GPU usando OpenGL ES 2.0 por uma questão de abrangência e compatibilidade com equipamentos disponíveis no mercado e facilmente acessíveis.

OpenGL e OpenCV possuem uma camada de interoperabilidade, mas de acordo com a documentação do OpenCV[37] no momento da realização deste trabalho, a camada de OpenGL só está funcional em computadores Desktop, ou seja, não há compatibilidade com OpenGL em dispositivos embarcados. Isso não implica que ambas as bibliotecas não funcionarão em conjunto, só significa que cada uma precisa ser chamada distintamente quando em operação em sistemas embarcados.

Um ponto importante que precisa ser observado quanto ao uso de OpenGL ES 2.0 em sistemas embarcados é que o mesmo depende tanto das bibliotecas específicas da API, quanto dos drivers e bibliotecas da GPU a ser utilizada. Caso contrário, os códigos não serão compilados, e muito menos executados.

No presente trabalho foi empregada a WandBoard Quad, que é apresentada em detalhes no próximo capítulo, e que possui uma GPU Vivante GC2000. Assim, para o desenvolvimento dos códigos e respectivas aplicações foi preciso baixar a SDK da Vivante, disponível em Freescale (2015).

[37] Documentação de Interoperabilidade disponível aqui:
http://docs.opencv.org/2.4/modules/core/doc/opengl_interop.html

4. Materiais e Métodos

No presente capítulo é apresentada a placa de desenvolvimento WandBoard Quad, utilizada para a execução dos programas e realização dos testes de desempenho, além de demais ferramentas computacionais utilizadas, como também detalhes técnicos a respeito da compilação destas ferramentas e de sua devida execução.

4.1. WandBoard Quad

Trata-se de um *kit* com o SoC i.MX6Q, que corresponde a um processador com quatro núcleos de ARM Cortex-A9 operando a frequência de 1,2 GHz, somado a 2 GB de memória RAM, além de uma GPU Vivante GC2000, compatível com OpenGL ES 2.0 e capaz de executar rotinas OpenCL EP 1.1.

Foi escolhida por se tratar de um dos sistemas embarcados mais poderosos disponíveis no mercado, no período de escrita e realização deste trabalho. Seu SoC é o mesmo presente na placa Sabre Lite, usada por Maghazeh et. al. (2013).

Além disso, também possui entrada para cartão de memória micros, conector SATA, uma porta USB 3.0 e outra porta USB On-The-Go, porta Serial, saída HDMI, saída e entrada de áudio, conexão de rede 10/100/1000 Ethernet, e já vem com dispositivos para conexão *Bluetooth* e *WiFi* embutidos. Como faz uso de um SoC que dissipa mais calor, esta versão vem com um dissipador de metal na placa de processamento. Na Figura 39 é mostrada a placa com perspectiva superior e inferor.

Esta placa não possui memória Flash integrada, fazendo com que seja obrigatório o uso de um cartão de memória devidamente configurado com o sistema operacional.

Como sistema operacional, foi utilizado o Ubuntu 12.04.3, com Kernel 3.0.35.

- **Preço:** US$ 129,00
- **Pagina do Projeto:** http://www.wandboard.org/ - Acesso em 01 de Março de 2016.

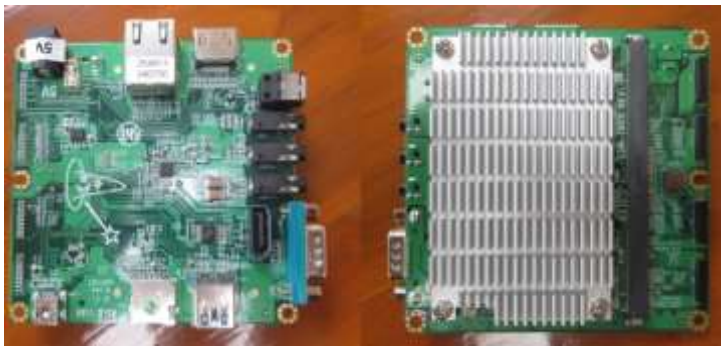


Figura 39 - Vistas inferior e superior da WandBoard Quad. Fonte: www.wandboard.org.

O fabricante do SoC utilizado, Freescale, disponibiliza para acesso público o esquemático interno de componentes do i.MX6Q, que pode ser visto na Figura 40. Interessante observar que o SoC também possui uma **Imaging Processing Unit**, que pode também ser utilizada em rotinas de processamento de imagens.

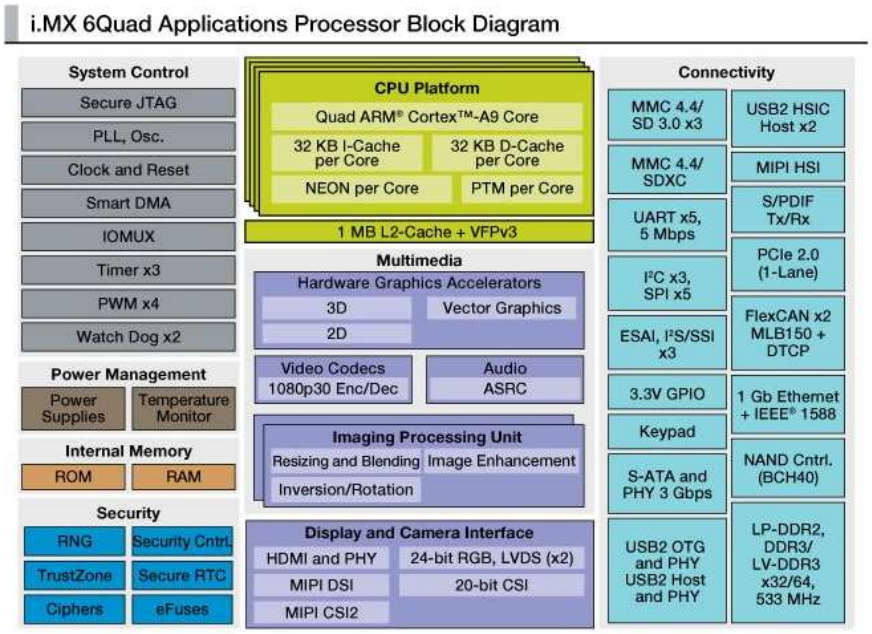


Figura 40 - Diagrama de blocos do SoC iMX6 Quad. Fonte: www.nxp.com.

4.1.1. GPU Vivante GC2000

De acordo com EMBEDDED.COM (2016), a GPU integrada ao SoC iMX6Q basicamente é composta por 3 unidades gráficas, a saber:

- **GC2000: OpenGL ES / OpenCL** – É o cerne principal, responsável por realizar as operações de OpenGL ES 2.0 em gráficos 3D, e possui suporte para OpenCL Embedded Profile 1.1.

- **GC355: OpenVG** – É um núcleo de GPU otimizado para acelerar operações com rotinas de gráficos vetorizados.
- **GC320: Composition** – Com os resultados gerados pelos processos realizados na GC2000 ou na GC355, a unidade GC320 trata exclusivamente de "juntar as peças", realizando a composição dos resultados das outras unidades para exibição do vídeo processado. Todavia, essa unidade também é capaz de processar gráficos 2D.

Dentre as unidades presentes, a que é diretamente utilizada é a GC2000, pois é ela a responsável pela execução das rotinas OpenGL ES 2.0 que farão o papel de processamento de imagens, ou seja, é na GC2000 que serão carregados o Vertex Shader e o Fragment Shader para realização do Filtro de Sobel. Nisso, o resultado do processo é enviado à GC320, que por sua vez cuida de exibir a janela de cenário no display gráfico. Na Figura 41 é mostrado o esquemático de operação da Vivante GC2000.

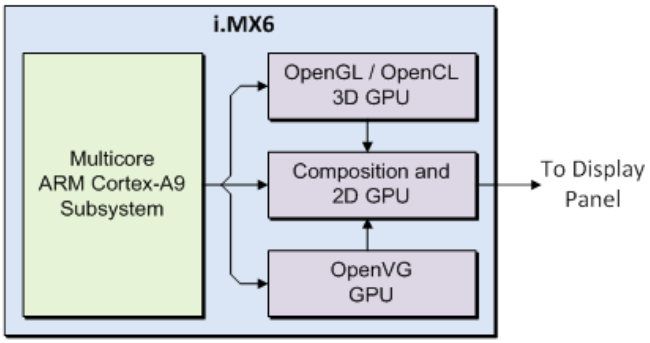


Figura 41 - Esquemático de Operação - Vivante GC2000. Fonte: www.embedded.com.

4.2. Benchmarks

Com base no trabalho de Roberts-Huffman e Hedge (2009), optou-se por realizar um *benchmark* inicial da WandBoard Quad, a título de estabelecer um critério inicial de desempenho com base nos *benchmarks* apresentados.

O *benchmark* inicial consistirá em uma bateria de testes com a aplicação das rotinas de *Dhrystone*, *Linpack* e *Whetstone*, para avaliar o desempenho da WandBoard Quad frente às variáveis trabalhadas em cada tipo de *benchmark*. Essas aplicações são chamadas de *benchmarks* sintéticos porque procuram mimetizar o comportamento do processador sob certos tipos de atividade.

4.2.1. Benchmark Dhrystone

Dhrystone[38] é um *benchmark* computacional sintético desenvolvido por Reinhold P. Weicker em 1984, com o objetivo de representar o poder computacional de um sistema com números inteiros. Como resultado do processo, o programa retorna o número de *Dhrystones* por segundo, que correspondem ao número de iterações executadas do código principal por segundo.

O arquivo fonte da aplicação *Dhrystone* pode ser obtido por meio do seguinte *link*:

http://www.xanthos.se/~joachim/dhrystone-src.tar.gz

Para baixar o pacote, pode-se utilizar o aplicativo *wget*, presente por padrão nas distribuições Linux utilizadas, da seguinte forma:

```
$ wget http://www.xanthos.se/~joachim/dhrystone-src.tar.gz
```

Após concluído o *download* do pacote, é necessário descompactar o arquivo, com o seguinte comando:

```
$ tar –xvzf dhrystone-src.tar.gz
```

Com isso, os arquivos do *benchmark* serão descompactados no diretório dhrystone-src. Os arquivos principais a serem compilados são **dhry21a.c**, **dhry21b.c** e **timers.c**.

O binário final foi compilado e executado tal como mostrado adiante:

- WandBoard i.MX6 Quad
  - Compilação do binário:

    ```
    $ gcc -O3 -mfpu=neon -ftree-vectorize dhry21a.c dhry21b.c timers.c -lm -o dhrystone
    ```

  - Execução do binário para teste:

    ```
    $ sudo ./dhrystone
    ```

    - Quantidade de rodadas: 100000000

[38] Maiores detalhes sobre o *benchmark* de *Dhrystone* podem ser encontrados neste *link*: http://www.roylongbottom.org.uk/classic.htm#anchorDhry

Os resultados obtidos com o *benchmark* de *Dhrystone* serão apresentados com resultados da média de 5 execuções do binário, ou seja, média dos resultados fornecidos por cada execução de *Dhrystone*.

4.2.2. Benchmark Linpack

O *benchmark Linpack*[39] foi desenvolvido por *Jack Dongarra* para o pacote *Linpack* de rotinas de álgebra linear. Essa aplicação acabou se tornando o principal *benchmark* para aplicações científicas em meados da década de 80. A versão original foi desenvolvida em FORTRAN, e posteriormente foi desenvolvida em Linguagem C.

A versão em Linguagem C do *benchmark Linpack* opera em matrizes com dimensões 100x100, com precisão de dados do tipo *double*. A título de curiosidade, é o *benchmark* utilizado pelo *site* TOP500 para avaliar os computadores mais rápidos do mundo.

O arquivo fonte do *benchmark* pode ser obtido através do seguinte *link*:

http://www.netlib.org/benchmark/linpackc.new

Para baixar o pacote, pode-se utilizar o aplicativo *wget*, presente por padrão nas distribuições Linux utilizadas, da seguinte forma:

```
$ wget http://www.netlib.org/benchmark/linpackc.new
```

Após concluído o *download* do pacote, é necessário renomear o arquivo, para deixa-lo com a extensão "*.c", com o seguinte comando:

```
$ mv linpackc.new linpack.c
```

Todas as rotinas necessárias para a realização do *benchmark* se encontram neste arquivo, e então o binário final foi compilado e executado tal como mostrado adiante:

- WandBoard i.MX6 Quad
  - Compilação do binário:

    ```
    $ gcc -O3 -mfpu=neon -lm -ftree-vectorize linpack.c -o linpack
    ```

  - Execução do binário para teste:

    ```
    $ sudo ./linpack
    ```

[39] Maiores detalhes a respeito do *benchmark Linpack,* e seus variantes para computação distribuída, podem ser vistos neste *link*: http://www.roylongbottom.org.uk/classic.htm#anchorLinp

- Dimensão da matriz: 200

Os resultados obtidos com o *benchmark* de *Linpack* serão apresentados com resultados da média de 5 execuções do binário, ou seja, média dos resultados fornecidos por cada execução de *Linpack*.

4.2.3. Benchmark Whetstone

O *benchmark Whetstone*[40] foi escrito por Harold Curnow, a agência britânica CCTA, com base no trabalho de Brian Wichmann do Laboratório Nacional de Física, da Inglaterra. Inicialmente, foi escrito em Algol em 1972, depois escrito em FORTRAN em 1973, e logo em seguida surgiu sua versão em Linguagem C. Basicamente, este *benchmark* avalia o desempenho do computador na operação de cálculos com ponto flutuante. Normalmente, o resultado do *benchmark* é fornecido em KWIPS, abreviação de *Kilo-Whetstones-Instructions-Per-Second*.

O arquivo fonte do *benchmark* pode ser obtido através do seguinte *link*:

http://netlib.org/benchmark/whetstone.c

Para baixar o pacote, pode-se utilizar o aplicativo *wget*, presente por padrão nas distribuições Linux utilizadas, da seguinte forma:

```
$ wget http://netlib.org/benchmark/whetstone.c
```

Todas as rotinas necessárias para a realização do *benchmark* se encontram neste arquivo, e então o binário final foi compilado e executado tal como mostrado adiante:

- WandBoard i.MX6 Quad
  - Compilação do binário:

    ```
    $ gcc -O3 -mfpu=neon -ftree-vectorize -o whetstone whetstone.c -lm
    ```

  - Execução do binário para teste:

    ```
    $ sudo ./whetstone 100000
    ```

Os resultados obtidos com o *benchmark* de *Whetstone* serão apresentados com resultados da média de 5 execuções do binário, ou seja, média dos resultados fornecidos por cada execução de *Whetstone* em cada máquina.

[40] Maiores detalhes sobre o *benchmark Whetstone* podem ser encontrados neste *link*: http://www.roylongbottom.org.uk/whetstone.htm.

4.3. Administração e controle do equipamento

Para controle e administração da WandBoard Quad utilizadas no trabalho, foi utilizada a forma de acesso remoto via SSH[41], juntamente com a conexão da saída de vídeo HDMI da placa em um display de vídeo compatível.

Em se tratando da versão de sistema operacional Ubuntu 12.04 instalada no cartão de memória da WandBoard, foi inicialmente necessário realizar acesso local para então proceder com os comandos de instalação dos pacotes para acesso remoto via SSH. Para instalar o pacote SSH bastou digitar o seguinte comando:

```
$ sudo apt-get install ssh
```

Feito isso, o Sistema cuida da instalação e configuração dos pacotes necessários. Com o término do processo, já é possível conectar e controlar a placa remotamente por meio do seguinte *login* de acesso via SSH:

- WandBoard Quad – Sistema Ubuntu versão 12.04
    - Conexão padrão: porta 22
    - Usuário: **linaro** / Senha: **linaro**.

Com a instalação do pacote SSH, também é possível realizar acesso para transferência de arquivos, e não somente para controle de terminal. Dessa forma, também foi possível acessar remotamente o sistema de arquivos da placa, possibilitando a transferência e edição dos arquivos de códigos necessários para execução dos testes.

4.4. Instalação e Compilação das Bibliotecas Necessárias

O compilador utilizado para compilar todos os programas escritos para testes é o GCC, para programas escritos em Linguagem C, e o G++, para compilar programas escritos em Linguagem C++. O sistema Linux Ubuntu 12.04 já vem com os compiladores GCC e G++ por padrão.

Infelizmente, não há por padrão a biblioteca OpenCV disponível para instalação através do utilitário *apt-get*, disponível em distribuições Linux derivadas do Debian. Dessa forma, a instalação tem que ser feita manualmente, envolvendo o *download*, extração, instalação de pacotes adicionais, e, por fim, a compilação da biblioteca.

[41] SSH é a abreviação de *Secure Shell*.

Para instalar a biblioteca OpenCV, é preciso antes baixar o pacote da biblioteca, que para fins do presente trabalho foi utilizada em sua versão 2.4.8. E para baixar o pacote, procede-se com o seguinte comando:

```
$ wget http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.8/opencv-2.4.8.zip/download
```

O comando acima irá fazer o *download* do arquivo **opencv-2.4.8.zip**. Os sistemas Linux das placas utilizadas não vieram com o pacote **unzip**, que permite descompactar arquivos com extensão "*.zip". Além disso, um tutorial específico para instalação da biblioteca para a Cubieboard[42] destaca a necessidade de instalar mais uma sequência de pacotes para evitar problemas futuros com incompatibilidades diversas. Dessa forma, segue o comando necessário para instalar pacotes adicionais:

```
$ sudo apt-get install cmake unzip libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev libjpeg-dev imagemagick
```

Em suma, os pacotes adicionais envolvem *codecs* para lidar com imagens, como por exemplo o pacote **imagemagick**. Terminado o processo de instalação, pode-se proceder com a descompactação do arquivo **opencv-2.4.8.zip**, com o seguinte comando:

```
$ unzip opencv-2.4.8.zip
```

Nisso, foi criado o diretório **opencv-2.4.8**, contendo todos os arquivos necessários para a compilação da biblioteca. Para então configurar e compilar a biblioteca, pode-se proceder com os seguintes comandos:

```
$ cd opencv-2.4.8

$ cmake -DCMAKE_BUILD_TYPE=RELEASE -DCMAKE_INSTALL_PREFIX=/usr/local -DENABLE_VFPV3=ON -DENABLE_NEON=ON -DWITH_TBB=ON -DBUILD_TBB=ON –DWITH_OPENMP=ON –DBUILD_OPENMP=ON -DBUILD_EXAMPLES=ON
```

[42] O tutorial completo para instalação da biblioteca OpenCV na placa Cubieboard pode ser encontrado no seguinte *link*: http://ele.aut.ac.ir/~mahmoudi/tutorials/installing-opencv-on-cubieboard/.

O comando **cmake** serve para preparar todos os pacotes e *scripts* necessários para compilar os pacotes correspondentes à biblioteca OpenCV, de acordo com os parâmetros passados, que determinam:

- -DENALBE_VFPV3=ON : Suporte à especificação VFPv3, de operações com ponto flutuante.
- -DENABLE_NEON=ON : Suporte ao coprocessador NEON.
- -DWITH_TBB=ON : Habilita o *download* e suporte à biblioteca TBB.
- -DBUILD_TBB=ON : Habilita a compilação e instalação da biblioteca TBB.
- -DWITH_OPENMP=ON : Habilita o *download* e suporte à biblioteca OpenMP.
- -DBUILD_OPENMP=ON : Habilita a compilação e instalação da biblioteca OpenMP.
- -DBUILD_EXAMPLES=ON : Habilita a compilação dos exemplos que acompanham a biblioteca OpenCV.

Dado o comando **cmake**, há um processo demorado para a configuração de todos os arquivos necessários para a compilação da biblioteca. Terminado o processo, pode-se proceder com a compilação e instalação, pelos comandos:

```
$ make
$ sudo make install
```

Com relação aos parâmetros passados ao **cmake** para instalação e compilação das bibliotecas TBB e OpenMP, vale ressaltar que a localização das bibliotecas compiladas pelo processo de instalação do OpenCV será no próprio diretório da biblioteca, sendo necessário passar parâmetros adicionais ao compilador GCC quando se fizer necessário seu uso.

E no que diz respeito ao compilador utilizado, GCC, há uma ampla documentação disponível, principalmente a respeito de *flags* específicas para compilação de programas para a arquitetura ARM, destacando o uso de unidade de ponto flutuante, especificação de arquitetura, dentre outras coisas[43].

Já em se tratando das bibliotecas para OpenGL, é possível proceder com sua instalação por meio do seguinte comando:

```
$ sudo apt-get install libgles2-mesa-dev libgl1-mesa-dev libgl1-mesa-glx libegl1-mesa libegl1-mesa-dev  libegl1-mesa-drivers mesa-utils-extra mesa-utils
```

[43] A lista completa de opções possíveis de utilizar com o compilador GCC para arquitetura ARM está disponível neste *link*: http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html

A biblioteca **libgles2-mesa-dev** é a principal delas, pois trata dos arquivos de desenvolvimento necessários para compilação de aplicações com OpenGL ES 2.0.

Como mencionado anteriormente, também é preciso ter a biblioteca para acesso aos recursos de driver da GPU utilizada, que no caso da WandBoard é a Vivante GC2000. As bibliotecas necessárias estão disponíveis no arquivo "**gpu_sdk_v1.00.tar.gz**" que pode ser baixado no site de arquivos da iMX6 da Freescale[44].

4.5. Medida de Tempo

Para medir o tempo gasto na execução de cada programa, foram adotadas as seguintes métricas de medida:

- **getTickCount()**[45], disponível na biblioteca OpenCV.
    - Retorna o número de pulsos de operação (*ticks*), depois de um dado evento. A diferença entre medidas de getTickCount() entre um ponto e outro de um programa pode ser utilizada para mensurar o tempo que rotinas tanto da biblioteca como de aspecto geral tomam para execução de tarefas.
- **getTickFrequency()**, disponível na biblioteca OpenCV.
    - Retorna o número de pulsos de operação (*ticks*) do sistema em segundos. Usada em conjunto com a **getTickCount**, permite então computar o tempo de execução de um dado trecho em escala de segundos.

Os programas escritos realizam tanto a parte de captura de imagem quanto processamento da imagem capturada. Como o foco é no **processamento de imagem**, foi computado apenas o tempo gasto nos trechos de código referentes ao processamento em si, ou seja, não foi considerado o tempo gasto para captura de quadros de imagem por câmera.

Então, com base nos recursos oferecidos por **getTickCount()** e **getTickFrequency()**, e com base tanto no algoritmo proposto por OpenCV (2016) e Freescale (2016), foi implementado

---

[44] Listagem de softwares para desenvolvimento com Freescale/NXP iMX6 - http://www.nxp.com/products/microcontrollers-and-processors/arm-processors/i.mx-applications-processors-based-on-arm-cores/i.mx-6-processors/i.mx6qp/i.mx-6-series-software-and-development-tool-resources:IMX6_SW

[45] A página do projeto OpenCV possui ampla documentação, inclusive sobre a função getTickCount(), getTickFrequency, dentre outras, disponível em: http://docs.opencv.org/modules/core/doc/utility_and_system_functions_and_macros.html.

o seguinte fluxo, mostrado na Figura 42, para realização da medida do tempo gasto nas rotinas de processamento de imagens, tanto em se tratando de rotinas executadas em CPU, como em rotinas executadas em GPU.

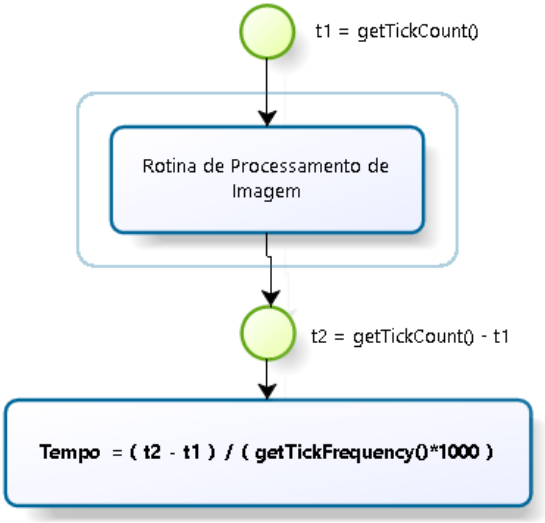


Figura 42 - Fluxo para medida de tempo em execução.

4.6. Execução

Para testar o desempenho da execução de código em GPU usando OpenGL ES 2.0 e usando CPU com OpenCV, tomou-se por base a aplicação de Filtros no contexto de processamento de imagens. Como dito na Introdução, Filtros são muito utilizados em processamento de imagens, sendo realizados por meio de convoluções. Maghazeh et. al (2013) comparou o desempenho entre uma Vivante GC2000 e uma GPU Tesla da Nvidia executando convoluções, pura e simplesmente. Todavia, neste trabalho as convoluções analisadas são realizadas sob a ótica de filtros aplicados em processamento de imagens.

Dentre os diversos filtros existentes, o Filtro de Sobel foi usado como base de operação a ser realizada por ambas as aplicações, escolhido pelo fato de ser um filtro comumente usado em rotinas de visão computacional, por permitir a identificação de bordas em imagens por cálculo de gradiente em cada pixel (GONZALEZ e WOODS, 2007).

Pelo fato de operar realizando a convolução de duas matrizes ao longo de todo um quadro de imagem: uma no eixo x, chamada por Gx, e outra no eixo y, Gy, conforme mostradas na Figura 43, é tido como um algoritmo computacionalmente pesado em detrimento da quantidade de operações matriciais realizadas. Existem mais outras duas matrizes de convolução para o Filtro

de Sobel, que tratam de operar os gradiantes na diagonal. Todavia, optou-se por usar as matrizes de gradientes no eixo x e no eixo y.

Na Figura 44 é apresentado um exemplo de aplicação do Filtro de Sobel, realizado sobre a imagem da Lena. Após a aplicação do filtro a imagem fica com os contornos destacados.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figura 43 - Matrizes de convolução em x e y para Filtro de Sobel. Fonte: http://homepages.inf.ed.ac.uk.

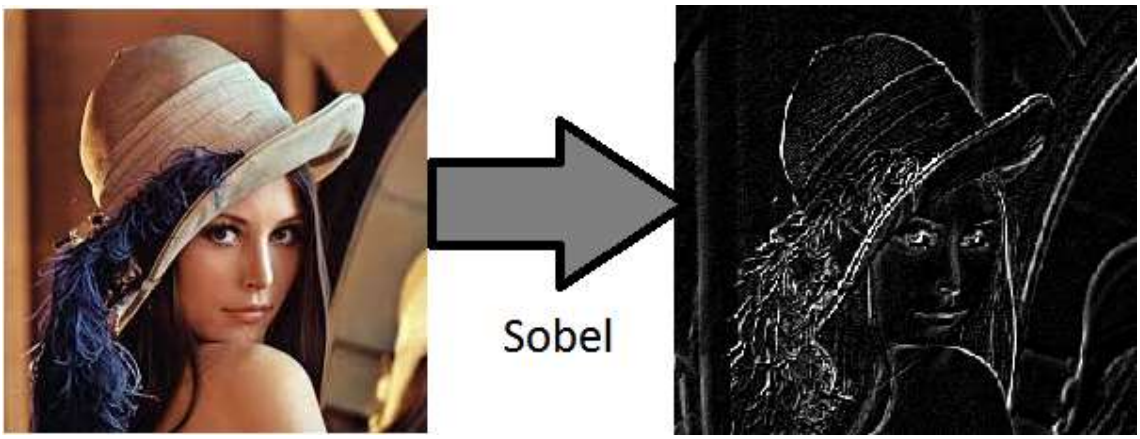


Figura 44 - Exemplo de aplicação do Filtro de Sobel.

A biblioteca OpenCV já possui uma estrutura bem fundamentada para a realização do algoritmo do Filtro de Sobel. Todavia, a execução desse filtro em GPU com OpenGL ES 2.0 não é trivial, o filtro não pode ser recursivo, devendo ser todo escrito de maneira procedimental, conforme mostrado em Freescale (2015).

Para demonstrar a aplicação modelo, foram preparados códigos para a execução de dois testes, a saber:

a. **Execução de código em CPU usando OpenCV** – Código apresentado no Apêndice A.1.
b. **Execução de código em GPU usando OpenGL ES 2.0** – Código apresentado no Apêndice A.4.

Cada aplicação exibe no Terminal o tempo gasto no processamento de cada quadro de imagem. Assim, são tomadas 10 medidas de tempo e calculada a média para cada programa. O resultado de cada rodada é apresentado no próximo capítulo, somado a considerações a respeito dos resultados obtidos.

Com relação à resolução trabalhada, ambos os programas desenvolvidos foram executados com as seguintes resoluções de captura, processamento e exibição de quadros de imagem: 320x240, 640x480, 800x600 e 1280x720 pixels.

A seguir são apresentados detalhes sobre estrutura, compilação e execução dos programas escritos para cada teste.

4.6.1. Código OpenCV em CPU

Com uso da biblioteca OpenCV, o código completo da aplicação é apresentado no Apêndice A.1, e o mesmo foi desenvolvido com base em conceitos apresentados por Bradski (2008), sendo compilado com o compilador G++.

Dessa forma, com os conceitos apresentados por Bradski (2008), o fluxograma de operação da aplicação com OpenCV executada em CPU é apresentado na Figura 45:

Figura 45 - Fluxo de operação - Aplicação OpenCV em CPU.

O fluxo apresentado na Figura 46 ficará em *loop* até que o programa seja encerrado, e a despeito de a captura de quadros de imagem contar no fluxo, ela foi desconsiderada no cálculo de tempo gasto na aplicação. Ou seja, apenas as etapas de Conversão de cor, aplicação do filtro de Sobel em X e em Y e formação da imagem com gradiente total é que fizeram parte da estrutura considerada no cálculo de tempo gasto para a aplicação com OpenCV na CPU.

A aplicação escrita em OpenCV faz uso do coprocessador NEON, e para compilar a aplicação os parâmetros de compilação são referenciados usando o utilitário **pkg-config**[46], capaz de informar a localização dos arquivos de cabeçalho e parâmetros de compilação para a biblioteca OpenCV. Demais parâmetros foram informados diretamente ao compilador, conforme apresentado adiante:

```
g++ openCVTest.cpp `pkg-config --cflags opencv` `pkg-config --libs opencv` -mfloat-abi=softfp -mfpu=neon -fPIC -fpermissive -O3 -fno-strict-aliasing -fno-optimize-sibling-calls -Wall -g -o openCVTest
```

Sobre os comandos informados ao compilador, segue uma listagem, considerando que as duas primeiras flags são as mais importantes, as demais foram apenas utilizadas a critério de otimização pelo compilador.

- **-mfloat-abi=softfp** – Comando que informa a geração de código usando instruções para processamento de ponto flutuante em hardware.
- **-mfpu=neon** – Comando que informa o uso de coprocessador NEON como unidade de ponto flutuante.
- **-fPIC** – Flag para código independente de posição.
- **-fpermissive** – Flag usada para utilizar algumas rotinas de GCC em G++.
- **-O3** – Ativa otimizações baixo-nível para o código em questão.
- **-Wall** – Indica que todas as mensagens de "aviso" do compilador devem ser mostradas.

Compilado o código, o mesmo pode ser executado com o seguinte comando:

```
$ ./openCVTest
```

[46] O pkg-config é um utilitário que auxilia na compilação de aplicações diversas, ao passo que fornece um mecanismo prático para referenciar cabeçalhos e bibliotecas necessárias para compilação. Mais detalhes em: https://www.freedesktop.org/wiki/Software/pkg-config/.

4.6.2. Código OpenGL em GPU

Por se tratar de um código bem mais complexo, que envolve a compilação e o link com bibliotecas do driver da GPU Vivante GC2000, foi empregado o arquivo **Makefile** apresentado no Apêndice A.5 como estrutura necessária para realizar a compilação do programa.

O código por completo, que é apresentado no Apêndice A.2, foi desenvolvido com base em conceitos vistos em Freescale (2015) e OBJC (2016), que basicamente forneceram um norte para esse tipo de aplicação ao apresentar a seguinte abordagem:

1. GPUs embarcadas compatíveis com OpenGL ES 2.0 são capazes de processar elementos em 3D, assim como a Vivante GC2000 da WandBoard iMX6Q.
2. Basta pegar um cubo em 3D, carregar uma imagem tal como se carregaria uma textura para dar forma ao cubo.
3. O cubo precisa ser centralizado em uma de suas faces, para exibição e processamento.
4. Carregada a imagem como textura, é possível usar programas Vertex Shader e Fragment Shader para realizarem operações sobre a imagem, em GPU.
5. Finalizado o processo, basta recuperar a imagem da GPU via OpenGL.

Na Figura 46 é apresentado um exemplo de cubo em 3D, renderizado em OpenGL, tendo como textura uma imagem de um par de aves.

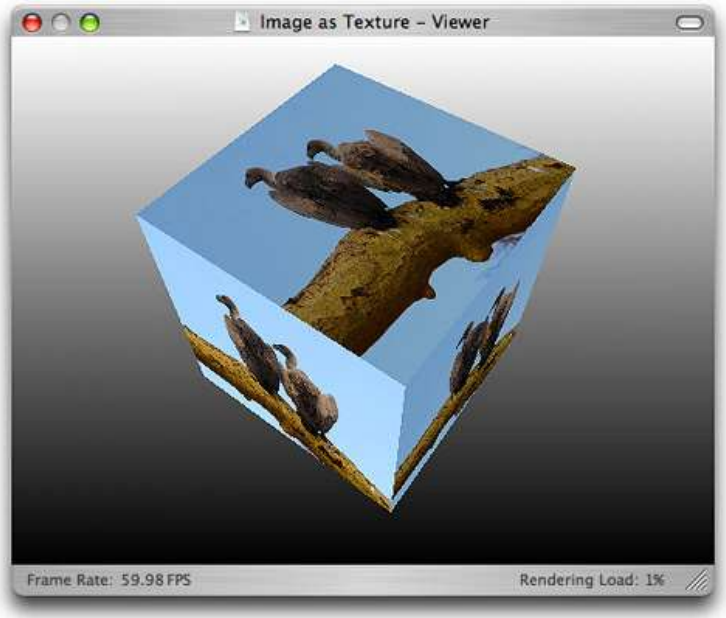


Figura 46 - Imagem como Textura em um Cubo com OpenGL. Fonte: www.apple.com.

Em suma, o código faz a mesma operação que a aplicação escrita com biblioteca OpenCV, com a exceção que a operação é realizada na GPU em OpenGL ES 2.0 por meio dos programas **Vertex Shader** e **Fragment Shader**.

Tomando por base os conceitos apresentados na nota de aplicação da Freescale (2015), foram escritos o Vertex Shader apresentado no Apêndice A.2, que serve para carregar as posições de vértices que serão utilizadas pelo programa também escrito Fragment Shader, apresentado no Apêndice A.3, que por sua vez irá pegar os dados de posição informados pelo Vertex Shader e realizar as seguintes rotinas, em ordem:

1. Conversão em escala de cinza – método **toGrayscale()** do Fragment Shader – É realizado pegando a média de escala de cores nos canais R (Vermelho), G (Verde) e B (Azul) do pixel.
2. Realiza convolução de maneira procedimental com as matrizes (kernels) do Filtro de Sobel, conforme apresentado no subcapítulo 3.6 e na Figura 44, nos eixos X e Y – método **doConvolution()** do Fragment Shader.
3. Os resultados dos processos de convolução em X e em Y são comparados frente a coeficientes pré-definidos experimentalmente, e com base nesse fluxo de comparação (realizado no método **main()** do Fragment Shader), o pixel é associado à cor branca ou à cor preta, criando assim as bordas na imagem conforme o Filtro de Sobel.

Inspirado no algoritmo apresentado por Freescale (2015), o código escrito para processamento de imagem em OpenGL faz uso de uma *thread* específica para capturar quadros de imagem, os quais são carregados em uma varíavel compartilhada com a rotina de renderização dos quados em OpenGL.

Como o tempo de captura de imagem não é considerado nos cálculos realizados, não houve penalização de desempenho quanto ao uso da *thread*. No código com OpenGL o tempo computado é apenas o de renderização dos quados de imagem do cubo 3D.

Dessa forma, o fluxo de operação do programa em OpenGL é mostrado na Figura 47, em que o programa inicia carregando todas as referências para operação OpenGL, passando pelo carregamento das instruções de programas Vertex Shader e Fragment Shader, e por fim o programa ficará em **loop** realizando a renderização do cubo com textura de imagem em GPU com base nos programas *shaders*, carregando em textura a imagem capturada pela Webcam através de uma *thread*.

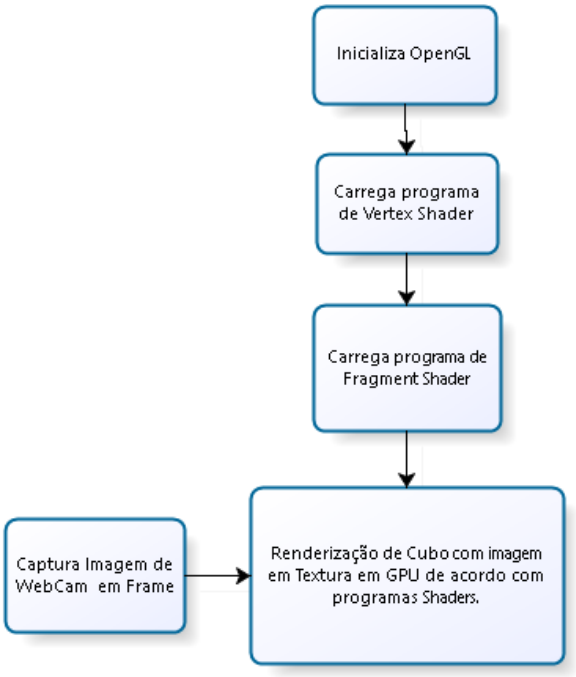


Figura 47 - Fluxo de Operação para programa de OpenGL em GPU.

No código apresentado no Apêndice A.4, observe a rotina **void render(float w, floag h)**, que é a rotina responsável por renderizar o cubo com imagem em textura. Dentro desta rotina, dentre outras chamadas, a de maior importância é a chamada à rotina **glTexImage2D()**, que irá pegar os dados do **Frame** de imagem capturado pela Webcam e carregar na forma de textura do cubo renderizado, que por sua vez passará pelo processo de execução dos códigos Vertex Shader e Fragment Shader.

Para compilar o código para usar o servidor gráfico X11, em operação na WandBoard Quad iMX6Q com Ubuntu 12.04, é preciso executar o seguinte comando:

```
$ make –f Makefile.x11
```

Antes de executar o programa, é preciso definir o display padrão empregado no sistema, o que pode ser feito com o seguinte comando em Terminal Linux:

```
$ export DISPLAY=:0.0
```

Todavia, é também preciso instanciar o módulo (driver) de suporte à GPU Vivante GC2000, definido pelo nome **galcore**, o que deve ser feito em Terminal Linux por meio do seguinte comando:

```
$ modprobe galcore
```

Instanciado o driver de suporte à GPU e definido o display padrão, com o término da compilação o código pode ser executado por meio do seguinte comando:

```
$ ./openGLTestCV
```

A cada quadro processado o programa exibe em Terminal o tempo gasto na renderização, tendo por base que a renderização é feita conforme os programas Fragment e Vertex Shaders carregados, este é então o tempo gasto pela GPU para computar as rotinas de Processamento de Imagens carregadas.

4.6.3. Recuperação de Imagem em GPU com OpenGL

Com relação ao uso de OpenGL para processamento de imagens em GPU, um ponto que merece destaque especial para medida e análise é também o processo reverso: o de recuperar a imagem processada em GPU via OpenGL para andamento em demais rotinas em OpenCV, na CPU. Esse processo é chamado na literatura por *readback*[47], que significa o processo de recuperar a informação do lugar onde foi armazenada.

A rotina-chave desse processo é a **glReadPixels()**[48], que irá ler os dados de pixels contidos no **FrameBuffer**, com base em coordenadas (x,y) passadas como parâmetros, e irá carregar esses dados de pixels em uma estrutura de memória.

A biblioteca OpenCV possui a estrutura Mat[49], que pode ser facilmente utilizada na glReadPixels para leitura dos pixels do quadro de imagem OpenGL.

Todavia, há uma diferença a ser considerada na forma como que quadros de imagem em OpenGL e OpenCV são organizados. Como mostrado na Figura 48 e visto em AIT (2016), os quadros de imagem OpenGL são organizados da esquerda pra direita, de baixo pra cima, enquanto que os quadros de imagem OpenCV são organizados da esquerda pra direita, e de cima pra baixo.

---

[47] Mais detalhes sobre o *readback* podem ser vistos em: http://gpgpu.org/developer/legacy-gpgpu-graphics-apis/faq#11

[48] Mais detalhes sobre a rotina glReadPixels() podem ser vistos neste link: https://www.khronos.org/opengles/sdk/docs/man/xhtml/glReadPixels.xml

[49] Mais detalhes sobre a estrutura de memória para frames de imagem Mat do OpenCV podem ser vistos aqui> http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html

Dessa forma, é preciso ter em mente esse arranjo para recuperar um quadro de imagem em OpenGL para OpenCV.

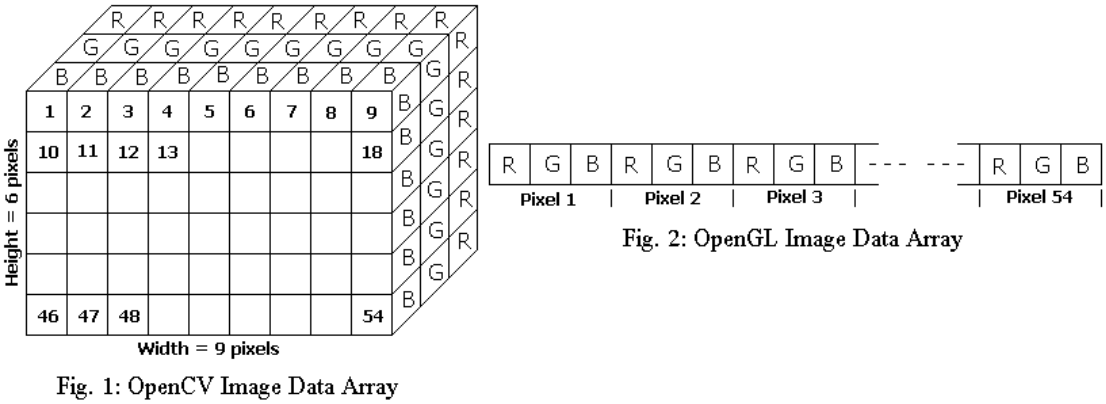


Figura 48 - Exemplo de representação de Pixels de imagem em OpenCV e em OpenGL. Fonte: http://vgl-ait.org

Copiada a imagem do FrameBuffer com o devido cuidado da ordem da informação, ela estará de "cabeça pra baixo", tal como mostrado na Figura 49, sendo necessário inverter horizontalmente a imagem por meio do método **flip()** da biblioteca OpenCV.

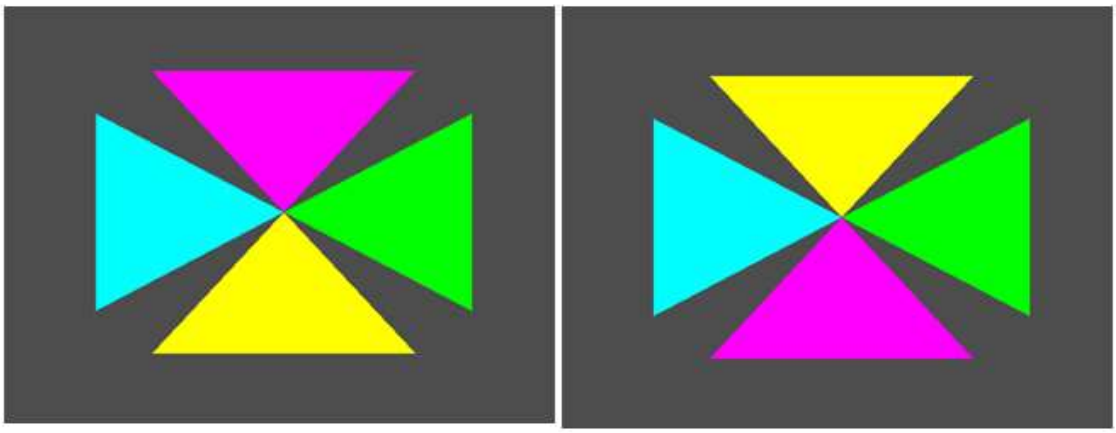


Figura 49 - Exemplo de processo de inversão (Flip) da imagem para recuperação de frame OpenGL. Fonte: http://vgl-ait.org.

De modo a avaliar o impacto provocado no tempo de execução do programa na recuperação dos quadros de imagem processados em GPU via OpenGL, a estrutura para medida de tempo apresentada no subcapítulo 3.5 também foi empregada no trecho de código responsável pela recuperação da imagem da GPU.

5. Resultados

Após a realização dos *benchmarks* de *Dhrystone*, *Whetstone* e *Linpack*, por influência do trabalho de Roberts-Huffman e Hedge (2009), as médias das 5 execuções de cada programa foram organizadas e agrupadas na Tabela 7 para a WandBoard Quad iMX6Q, e comparativamente foram adicionados à tabela os dados obtidos por Roberts-Huffman e Hedge (2009) para as plataformas BeagleBoard e Atom 330 Desktop.

Tabela 7 - Tabela com dados resultantes dos testes de *benchmark* para WandBoard..

Dados de desempenho	Dhrystone (VAX MIPS)	Whetstone (MIPS)	Linpack (MFLOPS)
WandBoard Quad	1638,98	1523,84	146,89
BeagleBoard	822	100	23,376
Atom 330 Desktop	1822	1667	933,638

Dessa forma, em se tratando dos benchmarks Dhrystone, a WandBoard Quad se sai duas vezes melhor que a BeagleBoard, e fica cerca de 200 pontos abaixo do Atom 330. No benchmark Whetstone, a WandBoard Quad é cerca de 15 vezes melhor que a BeagleBoard, e fica cerca de 100 pontos abaixo que o Atom 330. Só perdendo mais no benchmark Linpack, a WandBoard Quad se mostra uma plataforma embarcada com desempenho já bem semelhante à uma estação Desktop com processador Atom, por exemplo.

Considerando que o *benchmark* Linpack avalia cálculos em ponto-flutuante, essa pontuação não é necessariamente válida em um cenário de processamento de imagens, lembrando que os dados de pixels dos quadros de imagem são valores inteiros, normalmente com um intervalo de 256 valores para pixels de 8 *bits*.

Todavia, os resultados do *benchmark* Dhrystone são mais relevantes em uma aplicação de processamento de imagens, pois este *benchmark* avalia processamento com números inteiros (também considerado por *ponto-fixo*), que é o caso dos valores de *pixels* de imagens trabalhados nas aplicações de processamento de imagens. Dessa forma, a pontuação de 1638 da WandBoard Quad sendo bem próxima à do Atom 330 denota o potencial da plataforma embarcada para processamento de imagens.

Com a execução do código **openCVTest**, irá aparecer a janela mostrada na Figura 50 para exibir os quadros de imagem resultantes do processo de aplicação do Filtro de Sobel com as bibliotecas da suíte OpenCV.

Figura 50 - Execução do Filtro de Sobel com OpenCV na CPU.

O código foi projetado para exibir em console o tempo gasto no processo de cada quadro de imagem, conforme apresentado no subcapítulo 3.6.1. Os tempos de 10 quadros processados em sequência foram agrupados na Tabela 8 para análise, em escala de nanossegundos (ns), nas resoluções de 320x240, 640x480, 800x600 e 1280x720 pixels.

Tabela 8 - Tempos de execução para OpenCV na CPU.

OpenCV CPU				
Resolução	**320x240**	**640x480**	**800x600**	**1280x720**
Medidas de Tempo (ns)	2833,30	10976,90	15264,70	30735,70
	3239,83	10252,90	15273,30	31097,80
	5618,97	10257,90	15291,00	31188,10
	5027,33	10117,90	15294,30	31043,50
	3835,60	11618,20	15293,60	31014,30
	2641,23	10251,00	15300,70	31020,00
	2260,70	10332,60	15361,70	31284,60
	4430,77	11396,20	15354,00	31241,80
	3244,97	10328,10	15340,30	31106,90
	2273,10	11430,80	15816,00	31118,90
Tempo Médio Total (ns)	**3540,58**	**10696,25**	**15358,96**	**31085,16**

Já com a execução do código **openGLTestCV**, irá aparecer a janela mostrada na Figura 52, diferente da anteior em detrimento do uso de OpenGL para gerar a janela de exibição. Observe que a janela possui um preenchimento azul, e no interior está a imagem resultante do processo realizado na GPU. Isso se deve ao fato de que a imagem vista na Figura 51 é um cubo, com uma face centrada e carregada com a textura da imagem capturada e processada de acordo com os programas shaders.

Figura 51 - Execução do Filtro de Sobel com OpenGL na GPU.

Assim como na aplicação com OpenCV, este código foi projetado para exibir em console o tempo gasto no processo de **renderização em GPU** de cada quadro de imagem, conforme apresentado no subcapítulo 3.6.2. Os tempos de 10 quadros renderizados em sequência foram agrupados na Tabela 9 para análise, em escala de nanossegundos (ns), nas resoluções de 320x240, 640x480, 800x600 e 1280x720 pixels.

Tabela 9 - Tempos para Execução de rotinas OpenGL ES 2.0 em GPU.

OpenGL GPU				
Resolução	**320x240**	**640x480**	**800x600**	**1280x720**
Medidas de Tempo (ns)	1531,53	1478,60	1539,30	2703,43
	2157,73	2955,90	3484,33	4841,77
	1313,97	1459,53	1561,83	2446,80
	1478,13	1463,53	1663,43	2661,63
	1217,33	1482,33	3278,77	5014,50
	1515,13	2041,73	2097,77	2384,00
	1439,60	2008,97	1534,80	2737,37
	1345,13	1423,20	2960,57	4925,53
	1350,27	1437,30	1624,37	2396,87
	1202,00	1629,30	1611,63	2799,40
Tempo Médio Total (ns)	**1455,08**	**1738,04**	**2135,68**	**3291,13**

Para ter uma análise visual e quantitativa a respeito dos tempos gastos pela execução em CPU e em GPU, foi sintetizado o gráfico apresentado na Figura 52.

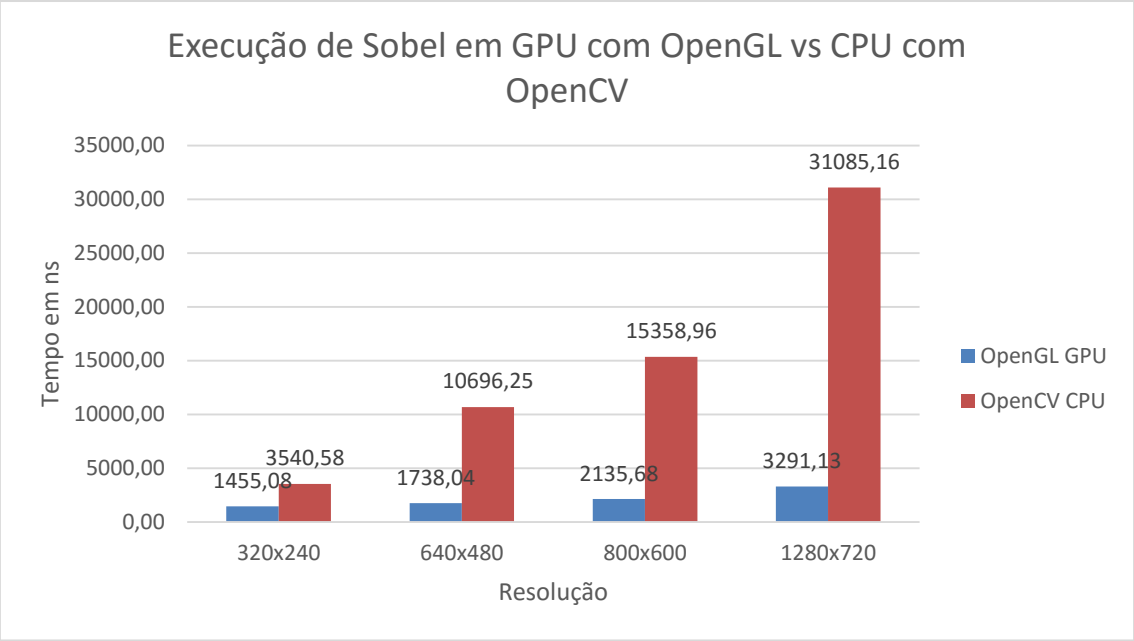


Figura 52 - Comparativo de execução entre OpenCV na CPU e OpenGL na GPU para Processamento de Imagem com Filtro de Sobel.

Comparando as medidas obtidas para as execuções do Filtro de Sobel em CPU e em GPU, foi possível observar que a execução em GPU apresentou ganhos de desempenho da ordem de 2.44, 6.16, 7.2 e 9.45 vezes para resolução de 320x240, 640x480, 800x600 e 1280x720, respectivamente, o que é melhor apresentado na Figura 53.

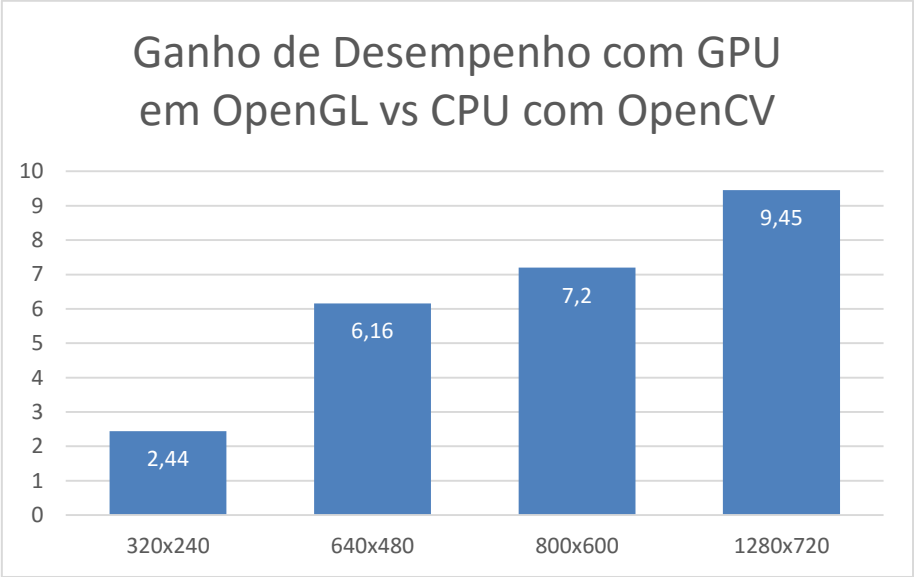


Figura 53 - Ganho de desempenho com uso de OpenGL para execução do Filtro de Sobel.

Ou seja, para uma resolução baixa, tal como 320x240 pixels, o ganho de desempenho foi pouco, comparativamente. Já com as resoluções de 640x480 e 800x600, relativamente próximas em dimensão, os ganhos de desempenho também foram próximos, da ordem de 6.1 e 7.2 vezes. A maior diferença ficou para a resolução de 720p, nativa para a câmera webcam utilizada, que ficou em 9.4 vezes para o código executado em GPU contra o código executado em CPU.

Todavia, só o tempo de processamento de imagem em GPU não é suficiente para concluir a eficiência dessa abordagem.

Assim como no caso do uso de DSP para processamento paralelo feito por Coombs e Prabhu (2011), a informação é enviada para a GPU, é processada, e depois deve ser recuperada. Esse tempo de recuperação é importante, e deve ser considerado na análise dessa abordagem.

Sendo assim, o código apresentado no Apêndice A.4 também conta com rotinas para recuperação dos quadros de imagem da GPU com OpenGL, como também está estruturado para avaliar o tempo gasto no processo de recuperação.

Em sequência ao processo de renderização dos quadros de imagem, os mesmos também foram recuperados da GPU com OpenGL para elementos de memória Mat equivalentes em OpenCV, e o tempo gasto na mesma sequência de 10 quadros renderizados mostrados na Tabela 9, os tempos gastos para recuperar esses mesmos quadros são apresentados na Tabela 10.

Tabela 10 - Tempos para recuperação (readback) de imagem de GPU em OpenGL para OpenCV

Recuperação de Quadros de Imagem de OpenGL para OpenCV				
Resolução	**320x240**	**640x480**	**800x600**	**1280x720**
Medidas de Tempo (ns)	230,73	1940,70	2051,57	3746,57
	226,30	1070,43	1727,07	3687,87
	226,87	1382,63	1939,87	4215,97
	205,27	1006,90	1968,37	4973,73
	204,87	1023,73	1979,53	5655,03
	207,80	1018,10	2483,20	3748,17
	238,37	940,37	1967,07	3769,27
	336,60	1820,00	1847,97	3718,97
	208,50	1052,40	1709,33	3714,90
	221,83	1050,03	1927,20	3747,37
Tempo Médio Total (ns)	**230,71**	**1230,53**	**1960,12**	**4097,79**

E para facilitar a observação dos resultados apresentados na Tabela 10, foi sintetizado o gráfico mostrado na Figura 54. Neste gráfico é possível ver que conforme a resolução da imagem aumenta, também aumenta o tempo gasto em recuperá-la da GPU. As imagens com resolução de 320x240 pixels foram as que apresentaram os menores tempos de recuperação, cerca de 230 ns,

já as imagens com resolução de 640x480 e 800x600 apresentaram tempos de recuperação da GPU da ordem de 1230 a 1960 ns, enquanto que imagens com resolução de 1280x720 apresentaram os maiores tempos de *readback*, da ordem de 4097 ns.

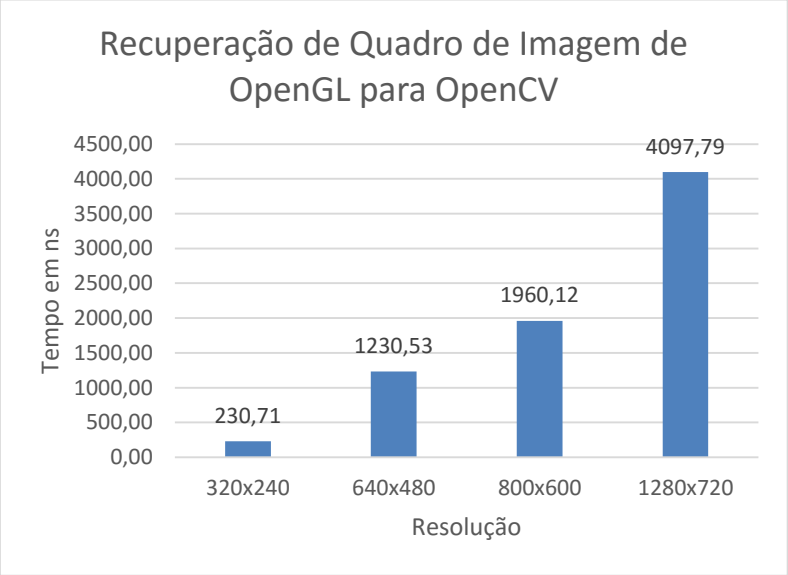


Figura 54 - Tempo de recuperação de imagem de GPU com OpenGL para OpenCV.

Com os dados de recuperação da imagem em GPU para OpenCV, foi montada a Tabela 11, que agrupa então o tempo total tomado para processamento/renderização da imagem em GPU, e sua recuperação para eventual processo em OpenCV.

Tabela 11 - Processamento + Recuperação de Imagem em OpenGL para OpenCV

Tempo Total - Processamento + Recuperação de Imagem em OpenGL para OpenCV				
Resolução	**320x240**	**640x480**	**800x600**	**1280x720**
Medidas de Tempo (ns)	1762,26	3419,30	3590,87	6450,00
	2384,03	4026,33	5211,40	8529,64
	1540,84	2842,16	3501,70	6662,77
	1683,40	2470,43	3631,80	7635,36
	1422,20	2506,06	5258,30	10669,53
	1722,93	3059,83	4580,97	6132,17
	1677,97	2949,34	3501,87	6506,64
	1681,73	3243,20	4808,54	8644,50
	1558,77	2489,70	3333,70	6111,77
	1423,83	2679,33	3538,83	6546,77
Tempo Médio Total (ns)	**1685,80**	**2968,57**	**4095,80**	**7388,92**

Com base nesses dados de tempo de execução total em GPU, podemos então efetuar mais uma análise sobre os tempos com relação à execução em CPU. Considerando os dados da Tabela 11 e os dados da Tabela 8, foi sintetizado o gráfico mostrado na Figura 55. Neste gráfico podemos

observar que as diferenças de tempo com a execução em GPU com OpenGL são diminuídas em decorrência da consideração, agora, do tempo gasto na recuperação da imagem da GPU.

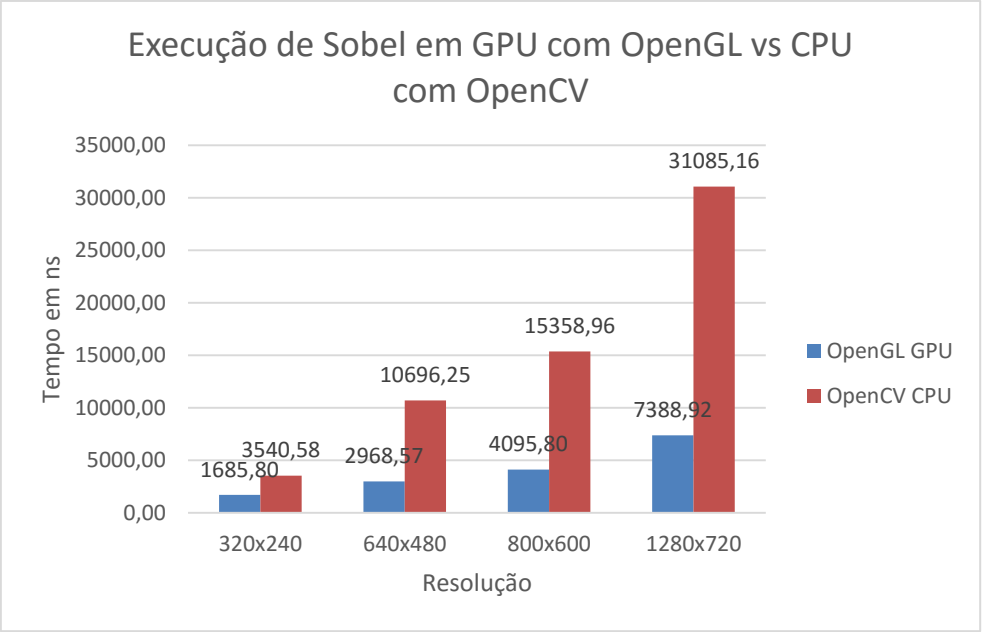


Figura 55 - Comparativo entre Execução do Filtro de Sobel em GPU com OpenGL + Tempo de recuperação da imagem Vs Execução na CPU com OpenCV.

Para se ter noção do impacto provocado pelo tempo necessário para recuperação dos quadros de imagem da GPU via OpenGL, os dados da Figura 55 foram transformados no gráfico da Figura 56. Com exceção da resolução 320x240, todas as demais resoluções tiveram o ganho de desempenho reduzido por praticamente a metade do apresentado no gráfico da Figura 53.

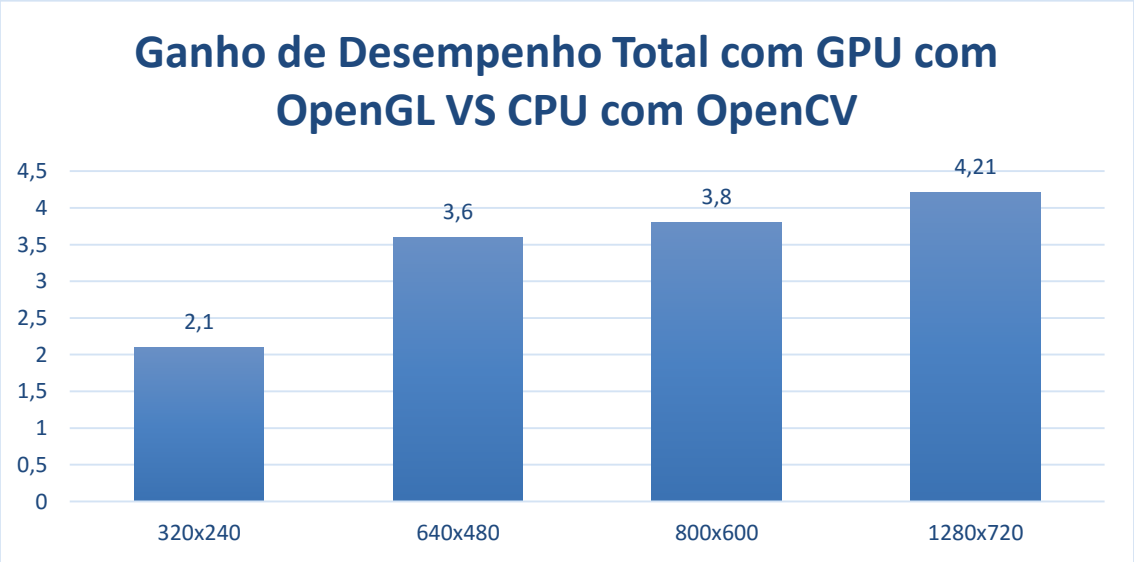


Figura 56 - Comparativo de Ganho de Desempenho Total com GPU usando OpenGL vs CPU com OpenCV.

Pulli et. Al. (2012) apresentaram uma proposta para aceleração de algoritmos de visão computacional usando OpenCV. Essa proposta foi elaborada em conjunto pela empresa It Seez[50], especializada em desenvolvimento e consultoria de projetos para Visão Computacional. A empresa It Seez fornece uma suíte de bibliotecas de OpenCV otimizadas por eles, tal como a biblioteca para aplicação do Filtro de Sobel, que foi utilizada no presente trabalho.

Acontece que a suíte desenvolvida pela It Seez é fechada e paga. Com base na Figura 57, a sua suíte de bibliotecas otimizadas para OpenCV apresenta um ganho de 9x sobre as rotinas tradicionais (públicas). Considerando o ganho de desempenho bruto, a proposta desse trabalho foi capaz de alcançar o mesmo índice de ganho de desempenho, com uma abordagem que será disponibilizada ao público.

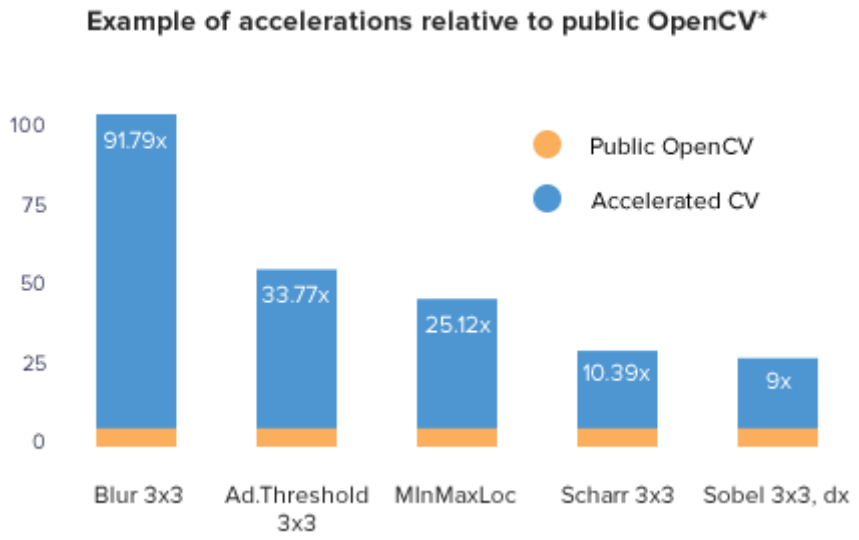


Figura 57 - Ganhos de desempenho com biblioteca de aceleração OpenCV. Fonte: itseez.com.

[50] Mais detalhes sobre a ItSeez podem ser vistos na página da empresa: http://itseez.com

6. Conclusões

Para a execução do Filtro de Sobel, considerando demais operações envolvidas tais como conversão de cores para escala de cinza, o código executado em GPU com uso de OpenGL ES 2.0 foi capaz de ter um desempenho bruto até 9x maior que o equivalente em CPU, todavia, há diferenças a serem consideradas nessa apresentação.

Como o foco do trabalho foi avaliar o desempenho em processamento de imagem na CPU e na GPU, não foi considerado o tempo gasto para a captura dos quadros de imagem do dispositivo de captura, que no presente trabalho foi uma câmera USB - Webcam.

O tempo gasto na captura dos quadros foi desconsiderado também pelo fato de que na aplicação com OpenCV o quadro é capturado a cada *loop* de execução do laço principal, enquanto que na aplicação com OpenGL ES 2.0 a captura é executada em uma *thread*. Ou seja, incluir o cálculo do tempo de captura para ambos os casos tornaria o código demasiado complexo.

Entretanto, ao analisar o desempenho da aplicação em GPU considerando o tempo necessário para recuperar a imagem processada em GPU via OpenGL para CPU, o ganho de desempenho caiu pela metade. Mesmo assim, continuou sendo um ganho de desempenho maior do que a execução das rotinas em CPU, numa escala aproximada de 4 vezes maior.

Tendo em vista que no *benchmark* de Whetstone, específico para cálculos com números inteiros, a WandBoard Quad ficou quase 200 pontos abaixo do resultado obtido pelo Atom 330, pode-se afirmar que com o ganho de desempenho obtido pela execução dos programas *shaders* em OpenGL ES 2.0 na GPU essa desvantagem é ultrapassada.

A aplicação escrita para se trabalhar com a GPU resultou em 670 linhas de código, considerando as rotinas de Vertex e Fragment Shaders, importantíssimas na operação-modelo para Filtro de Sobel, e sem a qual a aplicação perderia o sentido prático. Já a aplicação escrita em OpenCV e executada na CPU resultou em cerca de 70 linhas de código.

Sob a ótica de tamanho de código e aplicação para este caso em específico apresentado no trabalho, é possível dizer que o código executado na GPU foi *quase* 10x mais rápido que o equivalente em CPU, todavia, foi também *quase* 10x maior em tamanho de código.

A quantidade maior de código necessário para a operação com GPU usando OpenGL ES 2.0 se deve a todo um processo de inicialização dos componentes OpenGL tais como displays e janela, texturas, compilação dos programas Vertex Shader e Fragment Shader, vínculo entre variáveis dentre outros, o que resulta em um *overhead* de processamento não considerado no

cálculo de tempo de execução, até porque tais procedimentos são executados uma vez, na inicialização da aplicação.

O código escrito com OpenGL ES 2.0 para processamento de imagem em GPU apresentou um desempenho maior, mas é um código mais complexo que o equivalente com OpenCV executado em CPU.

O código escrito com OpenCV faz uso de bibliotecas incorporadas que facilitam a escrita e compreensão do código, e deixam o código principal da aplicação mais curto pelo uso das bibliotecas.

São muitas as possibilidades de aplicações fazendo uso da GPU com recursos da OpenGL ES 2.0, que é amplamente compatível com boa parte dos dispositivos embarcados com GPU disponíveis no mercado atualmente.

Todavia, é preciso ter um profundo conhecimento de rotinas OpenGL (quanto mais na vertente embarcada que possui algumas limitações, que é a OpenGL ES 2.0), programação de Shaders e domínio sobre a matemática acerca das rotinas de Processamento de Imagens, de modo que o desenvolvedor consiga escrever os programas Shaders capazes de realizar os processos desejados, lembrando que nesse cenário não é possível, ainda, o uso de recursão. Ou seja, os programas devem ser pensados em modo procedimental.

Mesmo em se tratando de uma aplicação com resultados satisfatórios, da ordem de um ganho de desempenho próximo de 10 vezes, uma documentação mais compreensível sobre o uso de OpenGL ES 2.0 voltado ao Processamento de Imagens em GPU é bem escassa, sendo um ponto-chave do presente trabalho o *Application Note* 4629 da Freescale[51]. Mesmo assim, o código apresentado naquele documento serviu apenas de inspiração em termos gerais e de algoritmos, pois faltam estruturas para seu correto funcionamento.

Uma boa base de código que facilitou a implementação do exemplo apresentado com execução em GPU foi a SDK fornecida pela Vivante, com códigos bem escritos e comentados, demonstrando aplicações de computação gráfica trabalhando com texturas, iluminação e reflexo, dentre outros. Unindo esta base com a do *Application Note* 4629 da Freescale, foi então possível escrever a aplicação-base para execução em GPU do trabalho.

[51] Application Note 4629 - Fast Image Processing with i.MX6 Series – Disponível em: http://cache.freescale.com/files/32bit/doc/app_note/AN4629.pdf

Um ponto que merece importante observação acerca do uso de OpenGL ES 2.0 é a necessidade de *drivers* GPU e suas bibliotecas de acesso, ou API. Sem os *drivers* ou bibliotecas, não é possível compilar, muito menos executar rotinas com OpenGL na GPU embarcada.

O destaque para esse ponto está no fato de que muitos fabricantes não liberam a totalidade de acesso aos recursos da GPU embarcada, ou, em outros casos, é de difícil acesso, sendo necessário acessar os sites do fabricante com *login* e eventualmente tendo que assinar algum contrato de confidencialidade.

A maior vantagem dos resultados obtidos com o trabalho está na demonstração do ganho de desempenho provocado pelo uso da GPU, aliado a um código bem documentado a respeito dos processos envolvidos para tornar isso possível.

Uma questão importante é: o ganho de desempenho vale o esforço? A despeito de todo o trabalho em criar um código para lidar com GPU, o ganho de desempenho justifica o esforço positivamente, ao passo que configurada e organizada a aplicação, a lógica de operação em *loop* contínuo é de semelhante modo tão simples quanto à aplicação criada para CPU.

E conforme mostrado no final do Capítulo 4, de Resultados, o trabalho foi realmente capaz de apresentar uma proposta funcional para aceleração de desempenho de algoritmos de visão computacional em sistemas embarcados, e em se tratando do algoritmo do Filtro de Sobel, equivalente em termos de desempenho com a proposta comercial apresentada pela empresa ItSeez.

7. Trabalhos Futuros

Todos os códigos e *scripts* desenvolvidos para o trabalho, apresentados na seção de Apêndices, serão disponibilizados ao público por meio do GitHub[52], tornando acessível as estruturas e rotinas aqui elaboradas. A página do GitHub já foi criada e pode ser acessada no seguinte endereço *web*: https://github.com/andrecurvello/mestrado_acc_gpu_embedded_opengl.

Isso tornará possível que estudantes e profissionais interessados encontrem uma estrutura-base bem organizada para fundamentar sua aplicação ou projeto de pesquisa, e assim continuar com os resultados obtidos por este trabalho.

De início, é possível já traçar uma série de novos estudos com base na execução dos códigos utilizando outros filtros de convolução em processamento de imagens. Para isso, basta somente alterar as matrizes de operação nos programas *shaders* carregados na GPU por meio de OpenGL ES 2.0.

Em um contato prévio com a comunidade por meio da rede social Facebook[53], foi possível identificar um interesse latente no desenvolvimento e aprimoramento das técnicas apresentadas no trabalho para as plataformas RaspberryPi e Toradex[54], o que se pretende executar dentro em breve na sequência desse trabalho.

A Raspberry Pi é um sistema embarcado voltado a Linux Embarcado, que já vendeu mais de 8 milhões de unidades (DailyMail, 2016). Um detalhe importante é que a RaspberryPi é dotada de uma GPU compatível com OpenGL ES 2.0.

Dessa forma, os códigos desenvolvidos para este trabalho serão adaptados para execução na GPU da Raspberry Pi, o que é uma abordagem interessante sob o aspecto de que há uma grande quantidade de Raspberry Pi em uso, seja por estudantes, faculdades, profissionais, que podem eventualmente fazer uso dos recursos aqui apresentados para otimização de desempenho de rotinas de processamento de imagens. E esse uso tem potencial impacto em aplicações de robótica, reconhecimento de gestos, dentre outras.

Com uma prerrogativa similar ao uso de sistemas embarcados de baixo cuso e baixo consumo de energia, o trabalho apresentado em Curvello et. al. (2015) demonstra o uso da

---

[52] GitHub é um serviço online para gestão de repositórios de códigos-fontes. Mais detalhes em: https://github.com/

[53] Facebook é uma rede social que permite a comunicação e interação entre pessoas do mundo todo. Mais detalhes em: http://www.facebook.com.br

[54] Toradex é uma fabricante Suíça de computadores-em-módulo (COM). Mais detalhes em: http://www.toradex.com.br

Raspberry Pi para transmissão de vídeo processado em OpenCV pela rede. Assim, é possível gerar mais um trabalho, conciliando as técnica de transmissão de vídeo pela rede, porém, agora considerando o processamento de imagens via OpenGL em GPU, e a recuperação da imagem para transmissão, por exemplo.

Dando continuidade no estudo de desempenho, o trabalho também será continuado na avaliação da execução de código em OpenGL ES 2.0 na GPU para aplicação de outros algoritmos de Processamento de Imagens, tais como Blur, Threshold, Color Tracking, dentre outros.

Os resultados apresentados no Capítulo 4 serão organizados para publicação no Workshop de Visão Computacional 2016 e demais congressos da área de Visão Computacional, juntamente com novos resultados obtidos com testes de mais aplicações e arquiteturas.

8. Referências

AIT, Computer Vision. *Saving Image from OpenGL to OpenCV*. Disponível em: <http://vgl-ait.org/cvwiki/doku.php?id=gltocv:main>. Acesso em 05 de Maio de 2016.

ALI, Akhtar. *Comparative study of parallel programming models for multicore computing*. Linköping University, 2013.

ARM. *ARMv8-A Architecture*. Disponível em: < http://www.arm.com/products/processors/armv8-architecture.php>. Acesso em 27 de Junho de 2016.

BRADSKI, Gary; KAEHLER, Adrian. *Learning OpenCV – Computer Vision with the OpenCV Library*. O'Reilly, 2008.

CUDA, Nvidia. *Parallel Programming and Computing Platform*. Disponível em: <http://www.nvidia.com/object/cuda_home_new.html>. Acesso em: 01 de Março de 2016.

COOMBS, Joseph; PRABHU, Rahul. *OpenCV on TI's DSP+ARM platforms: Mitigating the Challenges of porting OpenCV to embedded Platforms. White Paper at Texas Instruments*, 2011.

CURVELLO, André M L; LIMA, T. P. F. S. E. ; RODRIGUES, E. L. L. . A LOW COST EMBEDDED SYSTEM WITH COMPUTER VISION AND VIDEO STREAMING. In: XI Workshop de Visão Computacional, 2015, São Carlos. XI Workshop de Visão Computacional, 2015. p. 382-386

DAILYMAIL. *Raspberry Pi 3 adds Wi-Fi and Bluetooth: Firm unveils $35 board as it reveals it has become the UK's best-selling computer*. Disponível em: < http://www.dailymail.co.uk/sciencetech/article-3469209/Raspberry-Pi-UK-s-best-selling-computer-8-MILLION-boards-sold-four-years.html>. Acesso em 03 de Maio de 2016.

EMBEDDED.COM. *Auto Apps Accelerated By Triple-play Graphics Cores*. Disponível em: < http://www.embedded.com/print/4398965>. Acesso em 03 de Maio de 2016.

FREESCALE. 2D and 3D Graphics in Freescale Devices. Disponível em: < http://www.nxp.com/files/training/doc/dwf/DWF13_AMF_CON_T1025.pdf>. Acesso em 20 de Maio de 2016.

FREESCALE. *Application Note 4629 – Fast Image Processing with i.MX6 Series*. Disponível em: <http://cache.freescale.com/files/32bit/doc/app_note/AN4629.pdf>. Acesso em 01 de Março de 2016.

FREESCALE. *i.MX6 Series Software and Developtment Tools Resources.* Disponível em: < http://www.nxp.com/products/microcontrollers-and-processors/arm-processors/i.mx-applications-processors-based-on-arm-cores/i.mx-6-processors/i.mx6qp/i.mx-6-series-software-and-development-tool-resources:IMX6_SW>. Acesso em 01 de Março de 2015.

FREESCALE. *Software & Tools for SABRE Board for Smart Devices Based on the i.MX 6 Series*. Disponível em: <http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=RDIMX6SABREBRD&fpsp=1&tab=Design_Tools_Tab>. Acesso em 01 de Março de 2016.

GASTER, Benedict R.; HOWES, Lee; KAELI, David R.; MISTRY, Perhaad; SCHAA, Dana; *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2013

GNU, GCC Online Documentation. *Arm Options*. Disponível em: <http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html>. Acesso em 01 de Março de 2016.

GONZALEZ, Rafael C.; WOODS, Richard E. *Digital Image Processing*, 3rd Edition. Prentice Hall, 2007

KESSENICH, John; BALDWIN, Dave; ROST, Randi. *The OpenGL ES Shading Language*. Khronos Group, 2012.

KHRONOS. *OpenGL ES – The Standard for Embedded Accelerated 3D Graphics*. Disponível em: < https://www.khronos.org/opengles/2_X/>. Acesso em 01 de Março de 2016.

LANGBRIDGE, James A. *Professional Embedded ARM Development*. Wiley, 2014.

LINUX.COM. *Intro to Real-Time Linux for Embedded Developers*. Disponível em: < https://www.linux.com/blog/intro-real-time-linux-embedded-developers>. Acesso em 27 de Junho de 2016.

MAGHAZEH, Arian; BORDOLOI, Unmesh D.; ELES, Petru; PENG, Zebo. *General Purpose Computing on Low-Power Embedded GPUs: Has it Come of Age?* IEEE Xplore, 2013.

OBJC. *GPU-Accelerated Image Processing*. Disponível em: < https://www.objc.io/issues/21-camera-and-photos/gpu-accelerated-image-processing/>. Acesso em 01 de Março de 2016.

OPENCV. *OpenGL Interoperability.* Disponível em: <http://docs.opencv.org/2.4/modules/core/doc/opengl_interop.html>. Acesso em 01 de Março de 2016.

PULLI, Katri; Backsheev Anatoly; KORNYAKOV, Kirill; ERUHIMOV, Victor. *Realtime Computer Vision With OpenCV*. ACM Queue, 2012.

SIMMONDS, Chris. *A timeline for embedded Linux*. Disponível em: <http://events.linuxfoundation.org/sites/events/files/slides/csimmonds-embedded-linux-timeline-2013.pdf>. Acesso em 01 de Março de 2016.

THOUTI, Krishnahari; SATHE, S. R. *Comparison of OpenMP & OpenCL Parallel Processing Technologies*. International Journal of Advanced Computer Science and Applications, Vol. 3, No. 4, 2012.

TRISTAM, Waide; BRADSHAW, Karen. *Investigating the Performance and Code Characteristics of Three Parallel Programming Models for C++*. Rhodes University, 2010.

WILSON, Tom; DIPERT, Brian. *Embedded Vision on Mobile Devices – Opportunities and Challenges*. *Embedded Vision Alliance*, 2013. Disponível em: < http://www.eejournal.com/archives/articles/20130731-embvision/>. Acesso em 01 de Março de 2016.

YAGHMOUR, Karim; MASTERS, Jon; BEM-YOSSEF, Gilad; GERUM, Philippe. *Building Embedded Linux Systems, 2nd Edition*. O'Reilly, 2008.

Apêndice

A.1 - Código de captura de imagem e aplicação do filtro Sobel em OpenCV

```
#include "opencv2/opencv.hpp"
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;

int main(int, char**)
{
   //Inicializa objeto de captura da camera USB
   VideoCapture cap(0);
   //Verifica se ha acesso a camera
   if(!cap.isOpened())
      return -1;

   //Variaveis e demais objetos.
   double t1;
   double t2;
   Mat grad;
   Mat src;
   Mat src_gray;
   Mat sobel;
   Mat grad_x, grad_y;
   Mat abs_grad_x, abs_grad_y;
   int scale = 1;
   int delta = 0;
   int ddepth = CV_16S;

   //Definicao da janela de exibicao.
   namedWindow("Sobel OpenCV Neon",1);

   //Loop infinito de execucao.
   for(;;)
   {
      //Captura um quadro da camera
```

```
        cap >> src;
        //Referencia ao tempo de sistema neste ponto
        t1 = (double)getTickCount();
        cvtColor( src, src_gray, CV_BGR2GRAY );
        /// Gradiente em X
        Sobel( src_gray, grad_x, ddepth, 1, 0, 3, scale, delta, BORDER_DEFAULT );
        convertScaleAbs( grad_x, abs_grad_x );

        /// Gradiente em Y
        Sobel( src_gray, grad_y, ddepth, 0, 1, 3, scale, delta, BORDER_DEFAULT );
        convertScaleAbs( grad_y, abs_grad_y );

        /// Gradiente total (aproximado)
        addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad );
        //Referencia ao tempo de sistema neste ponto
        t2 = (double)getTickCount();
        //Imprime mensagem indicativa de tempo gasto a cada quadro pela diferenca de tempos.
        printf("Frame time: %gms\n", (t2-t1)/(getTickFrequency() * 1000.));
        //Exibe a imagem.
        imshow("Sobel OpenCV Neon - WandBoard Quad", grad);
        if(waitKey(30) >= 0) break;
    }
    return 0;
}
```

A.2 – Código de Vertex Shader para aplicação de Filtro de Sobel com OpenGL

```
uniform   mat4 g_matModelView;
uniform   mat4 g_matProj;

attribute vec4 g_vPosition;
attribute vec3 g_vColor;
attribute vec2 g_vTexCoord;

varying   vec3 g_vVSColor;
varying   vec2 g_vVSTexCoord;
```

```
void main()
{
   vec4 vPositionES = g_matModelView * g_vPosition;
   gl_Position  = g_matProj * vPositionES;
   g_vVSColor = g_vColor;
   g_vVSTexCoord = g_vTexCoord;
}
```

A.3 – Código de Fragment Shader para aplicação de Filtro de Sobel com OpenGL

```
#ifdef GL_FRAGMENT_PRECISION_HIGH
   precision highp float;
#else
   precision mediump float;
#endif

varying vec4 texColor;
varying vec2 g_vVSTexCoord;
uniform sampler2D s_texture;

mat3 kernel1 = mat3 (-1.0, -2.0, -1.0,
0.0, 0.0, 0.0,
1.0, 2.0, 1.0);

mat3 kernel2 = mat3 (-2.0, 0, 2,
0.0, 0.0, 0.0,
-1.0, 0.0, 1.0);

float toGrayscale(vec3 source) {
   float average = (source.x+source.y+source.z)/3.0;
   return average;
}
float doConvolution(mat3 kernel) {
   float sum = 0.0;
   float current_pixelColor = toGrayscale(texture2D(s_texture,g_vVSTexCoord).xyz);
   float xOffset = float(1)/1024.0;
   float yOffset = float(1)/768.0;
   float new_pixel00 = toGrayscale(texture2D(s_texture, vec2(g_vVSTexCoord.x-
xOffset,g_vVSTexCoord.y-yOffset)).xyz);
```

```
   float new_pixel01 = toGrayscale(texture2D(s_texture, vec2(g_vVSTexCoord.x,g_vVSTexCoord.y-
yOffset)).xyz);
   float new_pixel02 = toGrayscale(texture2D(s_texture,
vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y-yOffset)).xyz);
   vec3 pixelRow0 = vec3(new_pixel00,new_pixel01,new_pixel02);
   float new_pixel10 = toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x-
xOffset,g_vVSTexCoord.y)).xyz);
   float new_pixel11 =
toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x,g_vVSTexCoord.y)).xyz);
   float new_pixel12 =
toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y)).xyz);
   vec3 pixelRow1 = vec3(new_pixel10,new_pixel11,new_pixel12);
   float new_pixel20 = toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x-
xOffset,g_vVSTexCoord.y+yOffset)).xyz);
   float new_pixel21 =
toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x,g_vVSTexCoord.y+yOffset)).xyz);
   float new_pixel22 =
toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y+yOffset)).xyz);
   vec3 pixelRow2 = vec3(new_pixel20,new_pixel21,new_pixel22);
   vec3 mult1 = (kernel[0]*pixelRow0);
   vec3 mult2 = (kernel[1]*pixelRow1);
   vec3 mult3 = (kernel[2]*pixelRow2);
   sum= mult1.x+mult1.y+mult1.z+mult2.x+mult2.y+mult2.z+mult3.x+mult3.y+mult3.z;
   return sum;
}

void main() {
   float horizontalSum = 0.0;
   float verticalSum = 0.0;
   float averageSum = 0.0;
   horizontalSum = doConvolution(kernel1);
   verticalSum = doConvolution(kernel2);

   if((verticalSum > 0.2)|| (horizontalSum >0.2)||(verticalSum < -0.2)|| (horizontalSum <-0.2))
      averageSum = 0.0;
   else
      averageSum = 1.0;
   gl_FragColor = vec4(averageSum,averageSum,averageSum,1.0);
}
```

A.4 - Código de captura de imagem e aplicação do filtro Sobel com OpenGL

```
/*  Autor: André Márcio de Lima Curvello
 *  Código baseado no SDK da Vivante para iMX6
 *  Código realiza captura de imagens com OpenCV e
 *  aplica rotinas para processamento do Filtro de Sobel
 *  usando OpenGL.
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <assert.h>
#include <math.h>
#include <signal.h>
#include <pthread.h>
#include <sys/timeb.h>

#include "GLES2/gl2.h"
#include "GLES2/gl2ext.h"

#include "EGL/egl.h"

#include "FSL/fsl_egl.h"
#include "FSL/fslutil.h"

#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/legacy/legacy.hpp>
#include "opencv2/legacy/compat.hpp"

//Namespace para chamadas OpenCV com C++
using namespace cv;

#define TRUE 1
#define FALSE !TRUE

//Definicoes de variaveis EGL
EGLDisplay        egldisplay;
EGLConfig         eglconfig;
```

```
EGLSurface          eglsurface;
EGLContext          eglcontext;
EGLNativeWindowType eglNativeWindow;
EGLNativeDisplayType eglNativeDisplayType;

//Variavel de status
volatile sig_atomic_t quit = 0;

//Objeto de captura para camera com OpenCV
VideoCapture cap(0);
//Objetos para manipulacao de imagens com OpenCV
Mat edges;
Mat frame;

//Variaveis para resolucao da Webcam
int cap_width = 1280;
int cap_height = 720;
//Variavel para frame de captura de OpenGL
Mat frameOpenGL(cap_width, cap_height, CV_8UC3);

double timing;
double value;

//Thread e referencia para captura de imagens
pthread_t camera_thread;
int thread_id = 0;

//Variaveis de referencia para execucao de rotinas OpenGL
GLuint      g_hShaderProgram      = 0;
GLuint      g_hModelViewMatrixLoc = 0;
GLuint      g_hProjMatrixLoc      = 0;
GLuint      g_hVertexLoc          = 0;
GLuint      g_hVertexTexLoc       = 2;
GLuint      g_hColorLoc           = 1;


//-----------------------------------------------------------------------------------
// Desc: Programas Vertex e Shader para processamento em GPU com GLSL em OpenGL ES 2.0
//-----------------------------------------------------------------------------------
//Vertex Shader Padrao - Usado em todas as rotinas
```

```
const char* g_strVertexShader =
"uniform   mat4 g_matModelView; \n"
"uniform   mat4 g_matProj;      \n"
"                               \n"
"attribute vec4 g_vPosition;    \n"
"attribute vec3 g_vColor;       \n"
"attribute vec2 g_vTexCoord;    \n"
"                               \n"
"varying   vec3 g_vVSColor;     \n"
"varying   vec2 g_vVSTexCoord;  \n"
"                               \n"
"void main()                    \n"
"{                              \n"
"    vec4 vPositionES = g_matModelView * g_vPosition;\n"
"    gl_Position  = g_matProj * vPositionES;        \n"
"    g_vVSColor = g_vColor;                         \n"
"    g_vVSTexCoord = g_vTexCoord;                   \n"
"}                                                  \n";

//Fragment shader padrao - NAO faz modificacoes na imagem
const char* g_strFragmentShader =
"#ifdef GL_FRAGMENT_PRECISION_HIGH  \n"
"   precision highp float;          \n"
"#else                              \n"
"   precision mediump float;        \n"
"#endif                             \n"
"                                   \n"
"uniform sampler2D s_texture;       \n"
"varying vec3 g_vVSColor;           \n"
"varying vec2 g_vVSTexCoord;        \n"
"                                   \n"
"void main()                        \n"
"{                                  \n"
"    gl_FragColor = texture2D(s_texture,g_vVSTexCoord); \n"
"}                                  \n";

const char* plane_sobel_filter_shader_src =
"#ifdef GL_FRAGMENT_PRECISION_HIGH          \n"
"   precision highp float;                  \n"
"#else                                      \n"
```

```
"   precision mediump float;                \n"
"#endif                                     \n"
"                                           \n"
"varying vec4 texColor;                     \n"
"varying vec2 g_vVSTexCoord;                \n"
"uniform sampler2D s_texture;               \n"
"                                           \n"
"mat3 kernel1 = mat3 (-1.0, -2.0, -1.0,     \n"
"0.0, 0.0, 0.0,                             \n"
"1.0, 2.0, 1.0);                            \n"
"                                           \n"
"mat3 kernel2 = mat3 (-2.0, 0, 2,           \n"
"0.0, 0.0, 0.0,                             \n"
"-1.0, 0.0, 1.0);                           \n"
"                                           \n"
"float toGrayscale(vec3 source) {           \n"
"   float average = (source.x+source.y+source.z)/3.0;\n"
"   return average;                         \n"
"}                                          \n"
"float doConvolution(mat3 kernel) {         \n"
"   float sum = 0.0;\n"
"   float current_pixelColor = toGrayscale(texture2D(s_texture,g_vVSTexCoord).xyz);\n"
"   float xOffset = float(1)/1024.0;        \n"
"   float yOffset = float(1)/768.0;         \n"
"   float new_pixel00 = toGrayscale(texture2D(s_texture, vec2(g_vVSTexCoord.x-
xOffset,g_vVSTexCoord.y-yOffset)).xyz);\n"
"   float new_pixel01 = toGrayscale(texture2D(s_texture, vec2(g_vVSTexCoord.x,g_vVSTexCoord.y-
yOffset)).xyz);\n"
"   float new_pixel02 = toGrayscale(texture2D(s_texture,
vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y-yOffset)).xyz);\n"
"   vec3 pixelRow0 = vec3(new_pixel00,new_pixel01,new_pixel02);\n"
"   float new_pixel10 = toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x-
xOffset,g_vVSTexCoord.y)).xyz);          \n"
"   float new_pixel11 =
toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x,g_vVSTexCoord.y)).xyz);          \n"
"   float new_pixel12 =
toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y)).xyz);          \n"
"   vec3 pixelRow1 = vec3(new_pixel10,new_pixel11,new_pixel12);
\n"
```

```
"   float new_pixel20 = toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x-
xOffset,g_vVSTexCoord.y+yOffset)).xyz);\n"
"   float new_pixel21 =
toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x,g_vVSTexCoord.y+yOffset)).xyz);        \n"
"   float new_pixel22 =
toGrayscale(texture2D(s_texture,vec2(g_vVSTexCoord.x+xOffset,g_vVSTexCoord.y+yOffset)).xyz);\
n"
"   vec3 pixelRow2 = vec3(new_pixel20,new_pixel21,new_pixel22); \n"
"   vec3 mult1 = (kernel[0]*pixelRow0);                    \n"
"   vec3 mult2 = (kernel[1]*pixelRow1);                    \n"
"   vec3 mult3 = (kernel[2]*pixelRow2);                    \n"
"   sum= mult1.x+mult1.y+mult1.z+mult2.x+mult2.y+mult2.z+mult3.x+mult3.y+mult3.z;\n"
"   return sum;                                    \n"
"}                                                 \n"
"                                                  \n"
"void main() {                                     \n"
"   float horizontalSum = 0.0;                         \n"
"   float verticalSum = 0.0;                           \n"
"   float averageSum = 0.0;                            \n"
"   horizontalSum = doConvolution(kernel1);                \n"
"   verticalSum = doConvolution(kernel2);                  \n"
"                                                  \n"
"   if((verticalSum > 0.2)|| (horizontalSum >0.2)||(verticalSum < -0.2)|| (horizontalSum <-0.2))\n"
"      averageSum = 0.0;                               \n"
"   else                                           \n"
"      averageSum = 1.0;                               \n"
"                                                  \n"
"   gl_FragColor = vec4(averageSum,averageSum,averageSum,1.0);  \n"
"}\n";

//Referencias de posicoes para operacoes com Vertex.
float VertexPositions[]={
   /* Draw A Quad */
   /* Top Right Of The Quad (Top) */
   1.0f, 1.0f,-1.0f,
   /* Top Left Of The Quad (Top) */
   -1.0f, 1.0f, -1.0f,
   /* Bottom Right Of The Quad (Top) */
   1.0f,1.0f,1.0f,
   /* Bottom Left Of The Quad (Top) */
```

```
-1.0f,1.0f,1.0f,
/* Top Right Of The Quad (Bottom) */
1.0f,-1.0f,1.0f,
/* Top Left Of The Quad (Bottom) */
-1.0f, -1.0f,1.0f,
/* Bottom Right Of The Quad (Bottom) */
1.0f,-1.0f,-1.0f,
/* Bottom Left Of The Quad (Bottom) */
-1.0f,-1.0f,-1.0f,
/* Top Right Of The Quad (Front) */
1.0f,1.0f,1.0f,
/* Top Left Of The Quad (Front) */
-1.0f,1.0f,1.0f,
/* Bottom Right Of The Quad (Front) */
1.0f,-1.0f,1.0f,
/* Bottom Left Of The Quad (Front) */
-1.0f,-1.0f,1.0f,

/* Top Right Of The Quad (Back) */
1.0f,-1.0f,-1.0f,
/* Top Left Of The Quad (Back) */
-1.0f,-1.0f,-1.0f,
/* Bottom Right Of The Quad (Back) */
1.0f, 1.0f, -1.0f,
/* Bottom Left Of The Quad (Back) */
-1.0f,1.0f,-1.0f,

/* Top Right Of The Quad (Left) */
-1.0f,1.0f,1.0f,
/* Top Left Of The Quad (Left) */
-1.0f,1.0f, -1.0f,
/* Bottom Right Of The Quad (Left) */
-1.0f,-1.0f,1.0f,
/* Bottom Left Of The Quad (Left) */
-1.0f,-1.0f,-1.0f,
/* Top Right Of The Quad (Right) */
1.0f,1.0f,-1.0f,
/* Top Left Of The Quad (Right) */
1.0f,1.0f,1.0f,
/* Bottom Right Of The Quad (Right) */
```

```
    1.0f,-1.0f,-1.0f,
    /* Bottom Left Of The Quad (Right) */
    1.0f,-1.0f,1.0f

};

//Referencias de coordenadas para operacao com Vertex.
float VertexTexCoords[] = {

    /* Top Face */
    0.0f,0.0f,
    1.0f,0.0f,
    0.0f,1.0f,
    1.0f,1.0f,
    /* Bottom Face */
    1.0f,1.0f,
    0.0f,1.0f,
    0.0f,0.0f,
    1.0f,0.0f,

    /* Front Face */

    0.0f,0.0f,
    1.0f,0.0f,
    0.0f,1.0f,
    1.0f,1.0f,
    /* Back Face */
    1.0f,1.0f,
    0.0f,1.0f,
    1.0f,0.0f,
    0.0f,0.0f,
    /*left face*/
    0.0f,0.0f,
    1.0f,0.0f,
    0.0f,1.0f,
    1.0f,1.0f,

    /* Right face */
    0.0f,0.0f,
    1.0f,0.0f,
```

```
    0.0f,1.0f,
    1.0f,1.0f,

};

//Referencia de cores para operacoes com Vertex.
float VertexColors[] ={

    /* Red */
    1.0f, 0.0f, 0.0f, 1.0f,
    /* Red */
    1.0f, 0.0f, 0.0f, 1.0f,

    /* Green */
    0.0f, 1.0f, 0.0f, 1.0f,

    /* Green */
    0.0f, 1.0f, 0.0f, 1.0f,

    /* Blue */
    0.0f, 0.0f, 1.0f,1.0f,

    /* Blue */
    0.0f, 0.0f, 1.0f,1.0f,

    /* Red */
    1.0f, 0.0, 0.0f, 1.0f,
    /* Red */
    1.0f, 0.0, 0.0f, 1.0f,
    /* Blue */
    0.0f, 0.0f, 1.0f, 1.0f,
    /* Blue */
    0.0f, 0.0f, 1.0f, 1.0f,
    /* Green */
    0.0f, 1.0f, 0.0f, 1.0f,
    /* Green */
    0.0f, 1.0f, 0.0f, 1.0f,
    /* Red */
    1.0f, 0.0f, 0.0f,1.0f,
    /* Red */
```

```
    1.0f, 0.0f, 0.0f,1.0f,
    /* Green */
    0.0f, 1.0f, 0.0f,1.0f,
    /* Green */
    0.0f, 1.0f, 0.0f,1.0f,
    /* Blue */
    0.0f, 0.0f, 1.0f, 1.0f,
    /* Blue */
    0.0f, 0.0f, 1.0f, 1.0f,
    /* Red */
    1.0f, 0.0f, 0.0f, 1.0f,
    /* Red */
    1.0f, 0.0f, 0.0f, 1.0f,
    /* Blue */
    0.0f, 0.0f, 1.0f,1.0f,
    /* Green */
    0.0f, 1.0f, 0.0f, 1.0f,
    /* Blue */
    0.0f, 0.0f, 1.0f,1.0f,

    /* Green */
    0.0f, 1.0f, 0.0f, 1.0f


};

GLuint texture[1]; /* Armazenamento para 1 textura */


/* Funcao para carregar uma imagem capturada pela camera em textura */
int LoadGLTextures()
{
        cap >> frame;
        /* Alteracao de esquema de cores */
        cvtColor (frame, frame, CV_BGR2RGB);
        /* Criacao da textura */
        glGenTextures( 1, texture );
        /* Associacao da textura com elemento de textura */
        glBindTexture( GL_TEXTURE_2D, *texture );
        /* Criacao do elemento de Textura */
```

```
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, frame.cols, frame.rows, 0, GL_RGB,
GL_UNSIGNED_BYTE, frame.data);
        /* Filtragem linear */
        //Integer
        //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        //Float
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        printf("texture loaded and created successfully");

    return 1;
}

void render(float w, float h)
{
    static float fAngle = 0.0f;

    //Define a textura como sendo o novo quadro, recebido da captura de imagem da camera com
OpenCV.
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, frame.cols, frame.rows, 0, GL_RGB,
GL_UNSIGNED_BYTE, frame.data);

    float matModelView[16] = {0};
    matModelView[ 0] = cosf( fAngle );
    matModelView[ 2] = sinf( fAngle );
    matModelView[ 5] = 1.0f;
    matModelView[ 8] = sinf( fAngle );
    matModelView[10] = cosf( fAngle );
    matModelView[12] = 0.0f; //X
    matModelView[14] = -3.0f; //z
    matModelView[15] = 1.0f;

    // Construcao da matrix de projecao de perspectiva
    float matProj[16] = {0};
    matProj[ 0] = cosf( 0.5f ) / sinf( 0.5f );
    matProj[ 5] = matProj[0] * (w/h) ;
    matProj[10] = -( 10.0f ) / ( 9.0f );
    matProj[11] = -1.0f;
    matProj[14] = -( 10.0f ) / ( 9.0f );
```

```
// Limpeza de colorbuffer e depth-buffer
glClearColor( 0.0f, 0.0f, 0.5f, 1.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

// Definicao de parametros para execucao
glEnable( GL_DEPTH_TEST );
glEnable( GL_CULL_FACE );
glCullFace( GL_BACK );

// Define o programa shader.
glUseProgram( g_hShaderProgram );
glUniformMatrix4fv( g_hModelViewMatrixLoc, 1, 0, matModelView );
glUniformMatrix4fv( g_hProjMatrixLoc,      1, 0, matProj );

// Associa atributos de Vertex shader.
glVertexAttribPointer( g_hVertexLoc, 3, GL_FLOAT, 0, 0, VertexPositions );
glEnableVertexAttribArray( g_hVertexLoc );

glVertexAttribPointer( g_hColorLoc, 4, GL_FLOAT, 0, 0, VertexColors );
glEnableVertexAttribArray( g_hColorLoc );

glVertexAttribPointer( g_hVertexTexLoc, 2, GL_FLOAT, 0, 0, VertexTexCoords );
glEnableVertexAttribArray( g_hVertexTexLoc );

/* Seleciona textura */
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture[0]);

/* Drawing Using Triangle strips, draw triangle strips using 4 vertices */
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glDrawArrays(GL_TRIANGLE_STRIP, 4, 4);
glDrawArrays(GL_TRIANGLE_STRIP, 8, 4);
glDrawArrays(GL_TRIANGLE_STRIP, 12, 4);
glDrawArrays(GL_TRIANGLE_STRIP, 16, 4);
glDrawArrays(GL_TRIANGLE_STRIP, 20, 4);

// Limpeza
glDisableVertexAttribArray( g_hVertexLoc );
```

```
    glDisableVertexAttribArray( g_hColorLoc );
    glDisableVertexAttribArray( g_hVertexTexLoc );

    //Faz swap para atualizar display com nova informacao de textura
    eglSwapBuffers(egldisplay, eglsurface);
}


int init(void)
{
    //Configuracoes de atributos OpenGL ES - EGL.
    static const EGLint s_configAttribs[] =
    {
        EGL_RED_SIZE,       5,
        EGL_GREEN_SIZE,     6,
        EGL_BLUE_SIZE,      5,
        EGL_ALPHA_SIZE,     0,
        EGL_SAMPLES,        0,
        EGL_NONE
    };

    EGLint numconfigs;

    //Obtem display de execução do driver Vivante
    eglNativeDisplayType = fsl_getNativeDisplay();
    egldisplay = eglGetDisplay(eglNativeDisplayType);
    //Inicializa display
    eglInitialize(egldisplay, NULL, NULL);
    assert(eglGetError() == EGL_SUCCESS);

    //Associa api com OpenGL ES 2.0
    eglBindAPI(EGL_OPENGL_ES_API);

    //Inicializa configuracoes de instancia OpenGL ES 2.0
    eglChooseConfig(egldisplay, s_configAttribs, &eglconfig, 1, &numconfigs);
    assert(eglGetError() == EGL_SUCCESS);
    assert(numconfigs == 1);

    eglNativeWindow = fsl_createwindow(egldisplay, eglNativeDisplayType);
    assert(eglNativeWindow);
```

```
eglsurface = eglCreateWindowSurface(egldisplay, eglconfig, eglNativeWindow, NULL);

assert(eglGetError() == EGL_SUCCESS);
EGLint ContextAttribList[] = { EGL_CONTEXT_CLIENT_VERSION, 2, EGL_NONE };
//Inicia contexto de execucao vinculado ao display padrao de sistema - x11.
eglcontext = eglCreateContext( egldisplay, eglconfig, EGL_NO_CONTEXT, ContextAttribList );
assert(eglGetError() == EGL_SUCCESS);
//Associa janela de renderização opengl com display no contexto.
eglMakeCurrent(egldisplay, eglsurface, eglsurface, eglcontext);
assert(eglGetError() == EGL_SUCCESS);

{
   // Compilação dos shaders
   GLuint hVertexShader = glCreateShader( GL_VERTEX_SHADER );
   glShaderSource( hVertexShader, 1, &g_strVertexShader, NULL );
   glCompileShader( hVertexShader );

   // Verificia sucesso de compilação
   GLint nCompileResult = 0;
   glGetShaderiv( hVertexShader, GL_COMPILE_STATUS, &nCompileResult );
   if( 0 == nCompileResult )
   {
      printf("Vertex Shader compiler error!!!");
      char  strLog[1024];
      GLint nLength;
      glGetShaderInfoLog( hVertexShader, 1024, &nLength, strLog );
      //Imprime erro de compilação
      printf("%s",strLog);
      return FALSE;
   }

   GLuint hFragmentShader = glCreateShader( GL_FRAGMENT_SHADER );
   glShaderSource( hFragmentShader, 1, &plane_sobel_filter_shader_src, NULL );
   //glShaderSource( hFragmentShader, 1, &g_strTrackingShader, NULL );
   glCompileShader( hFragmentShader );

   // Verifica status de compilacao
   glGetShaderiv( hFragmentShader, GL_COMPILE_STATUS, &nCompileResult );
   if( 0 == nCompileResult )
```

```
{
   printf("Fragment Shader compiler error!");
   char  strLog[1024];
   GLint nLength;
   glGetShaderInfoLog( hFragmentShader, 1024, &nLength, strLog );
   //Imprime erro de compilação
   printf("%s",strLog);
   return FALSE;
}
// Associa os shaders individuais ao programa shader padrao
g_hShaderProgram = glCreateProgram();
glAttachShader( g_hShaderProgram, hVertexShader );
glAttachShader( g_hShaderProgram, hFragmentShader );

//Inicializacao de atributos antes do linking
glBindAttribLocation( g_hShaderProgram, g_hVertexLoc,   "g_vPosition" );
glBindAttribLocation( g_hShaderProgram, g_hColorLoc,    "g_vColor" );

printf("About to link shader...");
glBindAttribLocation( g_hShaderProgram, g_hVertexTexLoc,   "g_vTexCoord" );

//Link do vertex shader e do fragment shader
glLinkProgram( g_hShaderProgram );

//Verifica se o link ocorreu com sucesso
GLint nLinkResult = 0;
glGetProgramiv( g_hShaderProgram, GL_LINK_STATUS, &nLinkResult );
if( 0 == nLinkResult )
{
   char strLog[1024];
   GLint nLength;
   glGetProgramInfoLog( g_hShaderProgram, 1024, &nLength, strLog );
   printf("error linking shader");
   return FALSE;
}

// Get uniform locations
g_hModelViewMatrixLoc = glGetUniformLocation( g_hShaderProgram, "g_matModelView" );
g_hProjMatrixLoc      = glGetUniformLocation( g_hShaderProgram, "g_matProj" );
```

```
        glDeleteShader( hVertexShader );
        glDeleteShader( hFragmentShader );



        //gen textures
        /* Load in the texture */
        if (LoadGLTextures() == 0)
        {
            printf("error loading texture");
            return 0;
        }

        /* Enable Texture Mapping ( NEW ) */
        glEnable(GL_TEXTURE_2D);

    }



    return 1;
}

void Cleanup()
{

}

//Rotina para encerramento da instancia OpenGL ES 2.0
void deinit(void)
{
    printf("Cleaning up...\n");
    Cleanup();
    eglMakeCurrent(egldisplay, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
    assert(eglGetError() == EGL_SUCCESS);
    eglDestroyContext(egldisplay, eglcontext);
    eglDestroySurface(egldisplay, eglsurface);
    fsl_destroywindow(eglNativeWindow, eglNativeDisplayType);
    eglTerminate(egldisplay);
    assert(eglGetError() == EGL_SUCCESS);
    eglReleaseThread();
}
```

```
void sighandler(int signal)
{
   printf("Caught signal %d, setting flaq to quit.\n", signal);
   quit = 1;

}

//Rotina para thread de captura de quadros da camera USB
void *UpdateTextureFromCamera (void *ptr)
{
   while(1)
   {
      cap >> frame; // get a new frame from camera
      cvtColor(frame, frame, CV_BGR2RGB);
   }

}

int main (void)
{
   // Verifica se Webcam pode ser aberta
   if(!cap.isOpened()) {
      std::cout << "Falha na abertura da WebCam";
      return -1;
   }

   cap.set(CV_CAP_PROP_FRAME_WIDTH, cap_width);
   cap.set(CV_CAP_PROP_FRAME_HEIGHT, cap_height);


   //Inicializacao dos componentes
   signal(SIGINT, sighandler);
   signal(SIGTERM, sighandler);
   //Inicializacao da instancia de execucao OpenGL ES 2.0
   assert( init() );
   printf("OpenGL version supported by this platform (%s): \n", glGetString(GL_VERSION));
   //double fps = cap.get(CV_CAP_PROP_FPS); -- codigo pra FPS de captura
   //Impressao de detalhes sobre Width e Height de captura de imagem - padrao 640x480
   int CAM_WIDTH = cap.get(CV_CAP_PROP_FRAME_WIDTH);
```

```
int CAM_HEIGHT = cap.get(CV_CAP_PROP_FRAME_HEIGHT);
printf("Camera Capture Properties - Width - %d --- Camera Height - %d", CAM_WIDTH,
CAM_HEIGHT);

//Criacao da thread de captura de imagens da camera USB
pthread_create (&camera_thread, NULL, UpdateTextureFromCamera,(void *)&thread_id);
namedWindow("Sobel OpenCV OpenGL",1);
//glViewport(0, 0, cap_width, cap_height);
while (!quit)
{
    //Inicializacao de variaveis EGL para width e height de janela exibicao
    EGLint width = 0;
    EGLint height = 0;

    eglQuerySurface(egldisplay, eglsurface, EGL_WIDTH, &width);
    eglQuerySurface(egldisplay, eglsurface, EGL_HEIGHT, &height);

    //referencia de timing de sistema antes da renderizacao
    timing = (double)getTickCount();

    //Chamada de rotina de renderizacao
    render(cap_width, cap_height);

    //referencia de timing de sistema, apos renderizacao
    timing = (double)getTickCount() - timing;

    //calculo do tempo tomado para renderizacao
    value = timing/(getTickFrequency()*1000.0);

    //Impressao do tempo decorrido para renderizar janela
    printf( "\nFrame Time = %gms", value);

    //Calculo da rotina para obtencao da imagem da GPU com OpenGL:
    timing = (double)getTickCount();

    //Rotina de cópia do quadro de OpenGL no FB para estrutura Mat em OpenCV
    glReadPixels(0,0, frameOpenGL.cols, frameOpenGL.rows, GL_RGBA,
GL_UNSIGNED_BYTE, (GLubyte*)frameOpenGL.data);

    //Inversao da imagem recuperada de GPU com OpenGL
```

```
        flip(frameOpenGL, frameOpenGL, 0);

        //Diferenca de tempo
        timing = (double)getTickCount() - timing;

        //Calculo do tempo
        value = timing/(getTickFrequency()*1000.0);

        //Exibicao em Console.
        printf( "\nRecover GPU Frame Time = %gms", value);
        imwrite("Frame.png",frameOpenGL);
    }
    deinit();
    return 0;
}
```

A.5 - Arquivo Makefile para compilação

```
APPNAME        = openGLTestCV
DESTDIR        = .

# Make command to use for dependencies
CC = gcc
CXX = g++
AR = ar
LD = g++

TARGET_PATH_LIB = /usr/lib
TARGET_PATH_INCLUDE = /usr/include
COMMON_DIR=../common

BIN_TUT = $(DESTDIR)/$(APPNAME)

CFLAGS = -DDEBUG -D_DEBUG -D_GNU_SOURCE -mfloat-abi=softfp -mfpu=neon -fPIC -
fpermissive -O3 -fno-strict-aliasing -fno-optimize-sibling-calls -Wall -g `pkg-config --cflags opencv`

CFLAGS_TUT = $(CFLAGS) $(CFLAGS_EGL)

CFLAGS_TUT += \
        -DLINUX \
```

```
   -DEGL_USE_X11 \

CFLAGS_TUT += \
   -I. \
   -I$(TARGET_PATH_INCLUDE) \
   -I$(COMMON_DIR)/inc \

OBJECTS_TUT += \
   openGLTestCV.o \
   $(COMMON_DIR)/src/fsl_egl.o \
   $(COMMON_DIR)/src/fslutil.o \

ASSETS  = Texturing.bmp

DEPS_TUT = -lstdc++ -lm -lGLESv2 -lEGL -lX11 -ldl -Wl `pkg-config --libs opencv` #,--library-
path=$(TARGET_PATH_LIB),-rpath-link=$(TARGET_PATH_LIB)

$(BIN_TUT) : $(OBJECTS_TUT)
   @echo " LD " $@
   $(QUIET)$(CC) -o $(BIN_TUT) $(OBJECTS_TUT) $(DEPS_TUT)

%.o : %.c
   @echo " CC " $@
   $(QUIET)$(CC) $(CFLAGS_TUT) -MMD -c $< -o $@

%.o : %.cpp
   @echo " CXX " $@
   $(QUIET)$(CXX) $(CFLAGS_TUT) -MMD -c $< -o $@

clean:
   rm -f $(OBJECTS_TUT) $(OBJECTS_TUT:.o=.d) $(BIN_TUT)

install:
   cp -f $(APPNAME) $(ASSETS) $(DESTDIR)/.

-include $(OBJECTS_TUT:.o=.d)
```

A.6 – Script de Setup e Build de Binários

```
export DISPLAY=:0.0
make clean
make -f Makefile.x11
g++ openCV_Sobel.cpp `pkg-config --cflags opencv` `pkg-config --libs opencv` -mfloat-abi=softfp -mfpu=neon -fPIC -fpermissive -O3 -fno-strict-aliasing -fno-optimize-sibling-calls -Wall -g -o openCV_Sobel
```