UNVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
E INFORMÁTICA INDUSTRIAL

LUIZ HENRIQUE DUMA

UMA PROPOSTA DE MÉTODO PARA MELHORIA DE DESEMPENHO DO CODIFICADOR X264 BASEADA NA ANÁLISE DO ACESSO AO BARRAMENTO EXTERNO DE MEMÓRIA

DISSERTAÇÃO

CURITIBA
2011

LUIZ HENRIQUE DUMA

UMA PROPOSTA DE MÉTODO PARA MELHORIA DE DESEMPENHO DO CODIFICADOR X264 BASEADA NA ANÁLISE DO ACESSO AO BARRAMENTO EXTERNO DE MEMÓRIA

Dissertação apresentada ao Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de "Mestre em Ciências" - Área de Concentração: Telemática

Orientadora: Profª. Drª. Keiko Verônica Ono Fonseca

CURITIBA
2011

Dados Internacionais de Catalogação na Publicação

D897 Duma, Luiz Henrique
Uma proposta de método para melhoria de desempenho do decodificador X264 baseada na análise do acesso ao barramento externo de memória / Luiz Henrique Duma .— 2011.
82 f. : il. ; 30 cm

Orientador: Keiko Verônica Ono Fonseca.
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2011.
Bibliografia: f. 78-82.

1. Vídeo digital. 2. Decodificadores (Eletrônica) - Desempenho. 3. Teoria da codificação. 4. Processamento de imagens – Técnicas digitais. 5. Simulação (Computadores). 6. Engenharia elétrica – Dissertações. I. Fonseca, Keiko Verônica Ono, orient. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. III. Título.

CDD (22. ed.) 621.3

Biblioteca Central da UTFPR, Campus Curitiba

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CURITIBA

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Campus Curitiba

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL
CPGEI 22 anos

Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

Título da Dissertação Nº 573:

"Uma Proposta de Método para Melhoria de Desempenho do Codificador x264 Baseada na Análise do Acesso ao Barramento Externo de Memória"

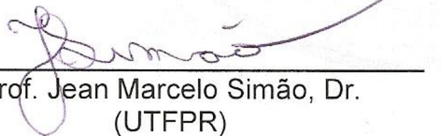
por

Luiz Henrique Duma

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM CIÊNCIAS – Área de Concentração: Telemática, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR – Campus Curitiba, às 14h do dia 26 de agosto de 2011. O trabalho foi aprovado pela Banca Examinadora, composta pelos professores:

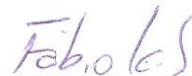
Profª. Keiko Verônica Ono Fonseca, Dr.
(Presidente – UTFPR)

Prof. Eduardo Todt, Dr.
(UFPR)



Prof. Jean Marcelo Simão, Dr.
(UTFPR)

Visto da coordenação:



Prof. Fábio Kurt Schneider, Dr.
(Coordenador do CPGEI)

Dedico este trabalho aos meus pais, Basilio e Iraci, que com muita dedicação e trabalho árduo, criaram as condições que me permitiram chegar até aqui.

AGRADECIMENTOS

Desejo expressar a minha gratidão a todas as pessoas, que direta ou indiretamente, contribuíram para o meu desenvolvimento pessoal e profissional.

Em especial, quero agradecer :

À minha esposa Debora, meu porto seguro em todos os momentos.

À professora Keiko, pela oportunidade de ingressar no programa de mestrado do CPGEI e pelas orientações que recebi desde a graduação.

Ao professor Alexandre de Almeida Prado Pohl e à equipe do laboratório LCD, pelos momentos de convivência e aprendizado que tivemos.

A todos do “CEFET-PR”, onde iniciei a minha preparação profissional e aos professores e funcionários do CPGEI onde pude desenvolver essa importante etapa da minha vida.

RESUMO

DUMA, Luiz Henrique. Uma proposta de método para melhoria de desempenho do codificador x264 baseada na análise do acesso ao barramento externo de memória. 82 f. Dissertação – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

A codificação de vídeo digital é um recurso essencial para a produção de vídeo para a Internet, canais de TV e outras mídias. Através da codificação é possível melhorar a utilização de recursos de armazenamento e da banda de transmissão e recepção de vídeo. Em sistemas embarcados, a limitação de recursos impacta no desempenho dos codificadores, como por exemplo, as câmeras de vídeo de telefones celulares. Este trabalho analisa o uso de técnicas para a diminuição de acesso à memória externa (RAM) especificamente para o codificador x264. Através do uso de ferramentas para *software profiling* e análise da performance do codificador a partir dos contadores de performance (HPC) disponíveis em muitos processadores modernos, foi possível estabelecer um método de análise de dados para direcionar a implementação do codificador para um melhor desempenho. Os resultados obtidos mostram uma melhora entre 16% e 18% no tempo de codificação em relação a um codificador não otimizado, mantendo-se os mesmos valores de qualidade de vídeo obtidos através de métricas objetivas.

Palavras-chave: H.264, codificador de vídeo, performance, barramento de memória, *software profiling*

ABSTRACT

DUMA, Luiz Henrique. Improving the performance of the x264 encoder based on the analysis of external memory bus access: a method proposal. 82 f. Dissertação – Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

This study attempts to systematize the use of techniques to reduce access to external memory (RAM) for the x264 encoder, as well the use of software profiling tools with focus on the usage of hardware performance counters (HPC), available in many modern processors. The results show up a reduction between 16% and 18% for execution time of the encoder, without noticeable changes on objective video quality metrics. Digital video coding is an essential resource to produce video for Internet, TV, and other media. Through video coding, it is possible to improve storage and bandwidth utilization for transmission and reception of video streams. On embedded devices, hardware resources impact on the encoder performance, for example, in video cameras of cellphones. This study analyzes the external memory access (RAM) at the x264 encoder implementation, aiming to identify ways to improve the encoding process performance. With software profiling tools and encoder performance analysis was possible to establish a data analysis method which results can be used to improve the overall encoder performance. The method implementation results show an improvement of 16% to 18% over a non-optimized encoder while keeping the same video quality measured from objective metrics.

Keywords: H.264, video encoder, performance, memory bus, software profiling

LISTA DE FIGURAS

FIGURA 1 – Diagrama em blocos do Intel *Core 2 Duo*. 18
FIGURA 2 – Diagrama em blocos do Intel *Core i7*. 18
FIGURA 3 – Execução de instruções no *pipeline* 19
FIGURA 4 – Execução de instruções sem *pipeline* 19
FIGURA 5 – Diagrama geral da *GPU* 25
FIGURA 6 – Diagrama em bloco da codificação H.264/*AVC* 26
FIGURA 7 – Matriz de Hadamard 28
FIGURA 8 – Varredura em zig-zag 29
FIGURA 9 – Exemplo do uso de instruções SSE (SIMD) 30
FIGURA 10 – Exemplo de execução do *gprof flat report* 38
FIGURA 11 – Exemplo de execução do *gprof call graph report* 39
FIGURA 12 – Exemplo de execução do *gprof annotated source report* 40
FIGURA 13 – Exemplo de execução do *perf-stat* 42
FIGURA 14 – Exemplo de execução múltipla do *perf-stat* 43
FIGURA 15 – Alguns eventos disponíveis no *perf* 43
FIGURA 16 – Alguns eventos disponíveis no *Oprofile* 44
FIGURA 17 – Exemplo de relatório gerado com o comando *perf-report –call-graph* 45
FIGURA 18 – Exemplo de relatório gerado com o comando *perf-annotate* 45
FIGURA 19 – Tipos de dados para os modos de acesso à memória 52
FIGURA 20 – Código-fonte da aplicação utilizada - 1/3 54
FIGURA 21 – Código-fonte da aplicação utilizada - 2/3 55
FIGURA 22 – Código-fonte da aplicação utilizada - 3/3 56
FIGURA 23 – Código-fonte da aplicação adaptada para a GPU 57
FIGURA 24 – Ciclos contabilizados para as operações de soma no *notebook1*. 59
FIGURA 25 – Ciclos contabilidados para as operações de soma no servidor1. 60
FIGURA 26 – Tempo de execução para cada um dos equipamentos. 60
FIGURA 27 – Imagens (quadros) obtidas dos vídeos utilizados 62
FIGURA 28 – Tipo de dado para o acesso à memória 63
FIGURA 29 – Fluxograma da análise e implementação 64
FIGURA 30 – Exemplo de execução do *perf-annotate* com o x264 65
FIGURA 31 – Exemplo de execução do *perf-report –call-graph* com o x.264 65
FIGURA 32 – Exemplo de execução do *perf-stat* com o x264 66
FIGURA 33 – Versão original SATD 8x4 67
FIGURA 34 – Implementação da otimização para SATD 8x4 67

LISTA DE TABELAS

TABELA 1 – Níveis hierárquicos típicos da memória. 15
TABELA 2 – Velocidade de memórias DDR 16
TABELA 3 – Características de alguns processadores 17
TABELA 4 – Largura de banda dos barramentos PCI e PCI-X 22
TABELA 5 – Taxa de transmissão básica por canal para o PCIe 23
TABELA 6 – Taxas de transferência para a interface PCIe versão 1.x 23
TABELA 7 – Especificação de algumas placas GPU NVidia 25
TABELA 8 – Redução média na taxa de bits para o H.264 29
TABELA 9 – Diferenças na sintaxe *assembly* para a arquitetura x86 32
TABELA 10 – Número de ciclos (x1000) para as principais rotinas no H.263 34
TABELA 11 – Acesso à memória das funções principais do codificador original 35
TABELA 12 – Performance do codificador com a otimização do *software* 37
TABELA 13 – Número total de ciclos da CPU para as etapas de otimização 37
TABELA 14 – Especificação da GPU utilizada 51
TABELA 15 – Performance no acesso à memória - *notebook1* 58
TABELA 16 – Performance no acesso à memória - servidor1 59
TABELA 17 – Características dos vídeos utilizados 62
TABELA 18 – Diretivas de compilação para controle das implementações 66
TABELA 19 – Versões dos codificadores 69
TABELA 20 – Dados coletados durante a execução dos codificadores 70
TABELA 21 – Dados coletados durante a execução dos codificadores 71
TABELA 22 – Dados coletados durante a execução dos codificadores 72
TABELA 23 – Variação nos eventos para o codificador - Osc x Oss 73
TABELA 24 – Características dos vídeos codificados 73

LISTA DE SIGLAS

AGP	Porta para Gráficos Acelerados (do original *Accelerated Graphics Port*)
API	Interface de Programação de Aplicativo (do original *Application Program Interface*)
APP	Processamento Paralelo Acelerado (do original *Accelerated Parallel Processing*)
AVC	Codificação Avançada de Vídeo (do original *Advanced Video Coding*)
BMPI	Impacto na Performance Devido a Predição de Desvio Incorreta (do original *Branch Misprediction Performance Impact*)
BP	Predição de desvio (do original *Branch Prediction*)
CABAC	Codificação Aritmética Binária Adaptativa ao Contexto (do original *Context-Adaptive Binary Arithmetic Coding*)
CAVLC	Codificação de Comprimento Váriavel Adaptativa ao Contexto (do original *Context-Adaptive Variable-Length Coding*)
CIF	Formato Intermediário Comum (do original *Common Intermediate Format*)
CPI	Taxa de Ciclos por Instrução Executada (do original *Clocks Per Instruction Retired Ratio*)
CPU	Unidade Central de Processamento (do original *Central Processor Unit*)
CUDA	Arquitetura de Dispositivo de Computação Unificada (do original *Compute Unified Device Architecture*)
DCT	Transformada Discreta de Cosseno (do original *Discrete Cosine Tranform*)
DDR	Taxa de Dados Duplicada (do original *Double Data Rate*)
DMI	Interface de Mídia Direta (do original *Direct Media Interface*)
DSP	Processador Digital de Sinais (do original *Digital Signal Processor*)
fps	Quadros por segundo (do original *Frames per seconds*)
FSB	Barramento Frontal (do original *Front Side Bus*)
GB	Gigabyte
Gbps	Gigabits por segundo
GCC	Conjunto de Compiladores GNU (do original *GNU Compiler Collection*)
GDDR	DDR para Interface Gráfica (do original Graphics Double Data Rate)
GHz	GigaHertz
(G)MCH	Controlador Central de Memória e Interface Gráfica (do original *Graphics Memory Controller Hub*)
GNU GPL	Licença Pública Geral GNU (do original GNU General Public License)
GPU	Unidade de Processamento Gráfico (do original *Graphics Processing Unit*)
HPC	Contador de Performance de *Hardware* (do original *Hardware Performance Counter*)
IEC	Comissão Eletrotécnica Internacional (do original *International Electrotechnical Commission*)
ISO	Organização Internacional de Padronização (do original *International Standards Organization*)
ITU	União Internacional de Telecomunicações (do original *International Tele-*

	communications Union)
JVT	Equipe Conjunta de Vídeo (do original *Joint Video Team*)
KB	Kilobyte
LAN	Rede de Área Local (do original *Local Area Network*)
LCD	Laboratório de Comunicação de Dados
LVDS	Sinalização Diferencial de Baixa Voltagem (do original *Low-Voltage Differential Signaling*)
MB	Megabyte
MCH	Controlador Central de Memória (do original *Memory Controller Hub*)
ME	Estimação de Movimento (do original Motion Estimation)
MHz	MegaHertz
MMX	Extensão Multimídia (do original *MultiMedia eXtension*)
MPEG	Grupo de Especialistas para Imagens em Movimento (do original *Moving Picture Experts Group*)
MSE	Erro Quadrático Médio (do original *Mean Square Error*)
OpenCL	Linguagem de Computação Aberta (do original *Open Computing Language*)
PAPI	API para Performance (do original *Performance API*)
PCH	Controlador Central da Plataforma (do original *Platform Controller Hub*)
PCIe	PCI Expressa (do original *PCI Express*)
PCI	Interconexão de Componentes Periféricos (do original *Peripheral Component Interconnect*)
PCI-SIG	Grupo de Interesse Especial PCI (do original *PCI Special Interest Group*)
PCI-X	PCI Extendida (do original *PCI-eXtended*)
PMU	Unidades de Monitoramento de Performance (do original *Performance Monitoring Units*)
PSNR	Relação Sinal Ruído de Pico (do original *Peak Signal-to-Noise Ratio*)
QDR	Taxa de Dados Quadruplicada (do original *Quad Data Rate*)
RAM	Memória de Acesso Aleatório (do original *Random Access Memory*)
RDO	Otimização Taxa-Distorção (do original *Rate-Distortion Optimization*)
SAD	Soma das Diferenças Absolutas (do original *Sum of Absolute Differences*)
SATD	Soma da Diferença Transformada Absoluta (do original *Sum of Absolute Transformed Difference*)
SDK	*Kit* de Desenvolvimento de *Software* (do original *Software Development Kit*)
SIMD	Instrução Única, Múltiplos Dados (do original *Single Instruction Multiple Data*)
SO	Sistema Operacional
SSE2	SSE 2 (do original *Streaming SIMD Extensions 2*)
SSE3	SSE 3 (do original *Streaming SIMD Extensions 3*)
SSE4	SSE 4 (do original *Streaming SIMD Extensions 4*)
SSE	Extensão SIMD para Fluxos de Mídia (do original *Streaming SIMD Extensions*)
SSIM	Similaridade Estrutural (do original *Structural Similarity*)
TB	Terabyte
USB	Barramento Serial Universal (do original *Universal Serial Bus*)
VCEG	Grupo de Especialistas em Codificação de Vídeo (do original *Video Coding Experts Group*)
VLC	Código de Comprimento Variável (do original *Variable-Length Code*)

SUMÁRIO

1 INTRODUÇÃO **12**
1.1 MOTIVAÇÃO 12
1.2 OBJETIVOS 12
1.3 ORGANIZAÇÃO 13
2 FUNDAMENTAÇÃO **14**
2.1 ARQUITETURA DE COMPUTADORES 14
2.1.1 Arquitetura Intel 14
2.1.2 *Pipeline* do processador e *Branch-prediction* 18
2.1.3 Barramento PCI Express 20
2.1.4 GPU 24
2.2 O PADRÃO H.264 26
2.3 TÉCNICAS PARA MELHORAR A PERFORMANCE DO *SOFTWARE* 29
2.3.1 Uso de Instruções SIMD 30
2.3.2 Aplicação de auto-vetorização 31
2.3.3 Bibliotecas especializadas 31
2.3.4 Uso de linguagem Assembly 32
2.4 MELHORIA DA PERFORMANCE DE CODIFICADORES DE VÍDEO 32
2.4.1 Melhoria no acesso à memória 33
2.4.2 Melhoria no acesso à memória e uso da linguagem *assembly* 33
2.4.3 Melhorias na manipulação de dados 34
2.4.4 Implementação de um codificador H.264 Baseline em DSP 36
2.5 *SOFTWARE PROFILING* 37
2.5.1 gprof 37
2.5.2 *Hardware Performance Counters* 38
2.5.3 Linux *Perf Tools* 42
2.6 QUALIDADE DO VÍDEO DIGITAL 46
2.7 CONCLUSÃO 47
3 DESENVOLVIMENTO **50**
3.1 EQUIPAMENTOS UTILIZADOS 51
3.2 PERFORMANCE NO ACESSO À MEMÓRIA 51
3.2.1 Implementação 52
3.2.2 Resultados 58
3.3 REDUÇÃO DO ACESSO À MEMÓRIA PARA O CODIFICADOR X264 61
3.3.1 Implementação 61
3.3.2 Resultados 68
3.4 CONCLUSÕES 74
4 CONCLUSÕES E TRABALHOS FUTUROS **76**
REFERÊNCIAS **78**

1 INTRODUÇÃO

A codificação de imagens e vídeos é um recurso importante para a redução na quantidade de informação gerada durante a captura desses sinais, sejam eles obtidos diretamente na forma digital ou digitalizados a partir de um sinal analógico. A compressão de dados obtida com a codificação permite melhorar a utilização dos recursos de armazenamento, principalmente em ambientes que produzem uma grande quantidade de imagens e vídeos e necessitam armazena-los, assim como permite a melhor utilização da banda de transmissão e recepção de fluxos de vídeo em redes de computadores ou nas transmissões via sinais de radiofrequência, como é o caso da TV Digital brasileira.

1.1 MOTIVAÇÃO

A baixa performance na codificação de vídeo para o padrão H.264 em um sistema embarcado equipado com um DSP *Dual-core*, em testes realizados no laboratório LCD na UTFPR, foi um dos motivadores para o inicio deste trabalho. No sistema utilizado, foi possível codificar um sinal de vídeo no padrão *Motion JPEG* a partir da entrada de vídeo do sistema, porém, não foi possível codificar o sinal no padrão H.264 devido à baixa performance na execução do codificador. Além disso, percebeu-se a necessidade de reduzir o uso de memória RAM e *Flash* requeridas pelo codificador, para que o mesmo pudesse ser utilizado no sistema embarcado.

1.2 OBJETIVOS

Este trabalho tem como objetivo propor um método para a melhoria no desempenho do codificador x264 a partir da redução nos acessos ao barramento externo de memória. Para atingir esses objetivos, serão utilizadas ferramentas de *software profiling* para a obtenção de informações sobre a execução do codificador no sistema, informações essas que serão usadas posteriormente para validar a técnica de otimização adotada. Como objetivo secundário, espera-se atingir uma boa relação entre o desempenho do codifica-

dor e o tamanho do binário produzido, afim de que esse possa ser utilizado em trabalhos futuros com sistemas embarcados.

1.3 ORGANIZAÇÃO

Esta dissertação está organizada em capítulos que abordam a fundamentação teórica e ferramentas de apoio para a análise da performance, aplicação das técnicas para redução do acesso à memória, análise dos resultados e conclusões.

A fundamentação teórica, desenvolvida no capítulo 2, revisa conceitos sobre a arquitetura de computadores, relacionados a temas abordados no capitulo 3, além de conceitos sobre o padrão de codificação H.264 e avaliação da qualidade de vídeo. Nesse capitulo também são apresentadas técnicas para otimização de aplicações em geral e técnicas especificas para codificadores de vídeo, além de ferramentas que podem ser utilizadas na análise da performance de *software*.

No capítulo 3 são propostas técnicas para a otimização do codificador e que depois são validadas via implementação e análise com as ferramentas de *software profiling*.

As conclusões gerais e pontos a serem abordados em trabalhos futuros são apresentados no capítulo 4.

2 FUNDAMENTAÇÃO

Este capítulo revisa alguns conceitos sobre arquitetura de computadores com o objetivo de identificar pontos que podem impactar na performance das aplicações e devem ser considerados durante o projeto e especificação dos sistemas. Também são apresentadas técnicas que permitem melhorar a performance das aplicações tanto de uso geral quanto codificadores de vídeo, além de ferramentas que podem auxiliar durante a análise e identificação de possíveis pontos de melhoria. O padrão de codificação de vídeo H.264, implementado no codificador x264, e conceitos sobre qualidade de vídeo digital também serão abordados.

2.1 ARQUITETURA DE COMPUTADORES

Essa seção aborda assuntos relacionados ao *hardware* dos computadores de uso geral, como a comunicação do processador e os periféricos na arquitetura Intel, o processamento paralelo obtido com o uso de *pipelines*, os barramentos para interconexão de periféricos e as placas GPU.

2.1.1 ARQUITETURA INTEL

Ao longo dos anos, os processadores evoluíram em capacidade de processamento, melhorando principalmente o processamento em *pipeline* e o processamento superescalar (Os conceitos de *pipeline* e processamento superescalar são abordados na sub-seção 2.1.2). Os sistemas de memórias, por sua vez, evoluíram focando principalmente o aumento da capacidade de armazenamento, gerando uma distorção entre a velocidade das memórias e a dos processadores. Desse modo, o processador precisa aguardar muitos ciclos até que tenha acesso aos dados requisitados à memória (TANENBAUM, 2007).

Uma solução para o problema de velocidade de acesso é incluir as memórias dentro dos *chips* dos processadores mas isso os torna maiores e mais caros. Para resolver o problema de custo, adiciona-se uma memória de menor capacidade ao processador.

Os dados e/ou instruções acessados são armazenados nessa memória intermediária com o objetivo de reusá-los, evitando assim sucessivos acessos ao barramento externo. Essa memória "interna" é chamada de memória *cache*.

A operação das memórias *cache* se fundamenta no princípio da localidade, tanto temporal quanto espacial. Em linhas gerais, o principio da localidade espacial leva em consideração o fato de que durante a execução de um aplicativo, o acesso à memória não é totalmente aleatório e é muito provável que após o acesso a um endereço X, o próximo acesso ocorra na vizinhança de X.

Esse conceito é válido tanto para a seção de dados quanto para a seção de instruções de uma aplicação. Quando as instruções e dados utilizam a mesma *cache*, ela é chamada de *cache* unificada e é mais simples de ser projetada. A *cache* dividida, conhecida como arquitetura Harvard, gerencia separadamente os acessos a dados e instruções (TANENBAUM, 2007).

Internamente, as *caches* são organizadas em linhas de *cache*, que possuem um tamanho fixo. Assim, durante a tentativa de acesso a um determinado endereço, o endereço solicitado e seus vizinhos são copiados para a *cache*, levando em consideração o princípio da localidade espacial.

A localidade temporal está relacionada com o efetivo acesso ao dado armazenado em *cache*, acesso esse que pode ocorrer por exemplo na execução de instruções dentro de um laço. Os algoritmos de substituição da *cache* exploram o conceito da localidade temporal para determinar quais dados devem ser descartados e substituídos por dados mais atualizados (TANENBAUM, 2007).

Tabela 1: Níveis hierárquicos típicos da memória.

Nível	1	2	3	4
Tipo	registradores	*cache*	Memória principal (RAM)	Armazenamento
Tamanho tipico	< 1 KB	< 16 MB	< 512 GB	> 1 TB
Tempo de acesso [ns]	0,25-0,5	0,5-25	50-250	5.000.000
Gerenciado por	compilador	*hardware*	SO	SO/usuário

Fonte: Adaptado de (HENNESSY; PATTERSON, 2006) [Figure C.1]

Na tabela 1, o tempo de acesso de cada tipo de memória segue na ordem inversa da capacidade de armazenamento do dispositivo. Os registradores por exemplo, podem ser usados para armazenamento do valor de variáveis ou do resultado intermediário de uma operação matemática que está sendo executada no momento. O uso desse tipo de

memória é gerenciado pelo compilador. O desenvolvedor pode, durante a implementação, indicar que determinada variável deve ser armazenada em registradores, mas isso depende da linguagem de programação utilizada e normalmente cabe ao compilador decidir se é possível atender a solicitação do desenvolvedor naquele trecho de código. O gerenciamento do uso da memória *cache* é feito pelo controlador de *hardware* interno ao processador, responsável por manter atualizado o conteúdo da mesma. O acesso à memória RAM é gerenciado pelo Sistema Operacional, durante a inicialização da aplicação ou quando esta faz uma solicitação de alocação dinâmica de memória durante a execução.

Tabela 2: Velocidade de memórias DDR

Tipo	Frequência Operação [MHz]	Frequência Transferência [MHz]	Taxa Transferência [GB/s]	Largura Barramento [Bits]
DDR2	200	400	3,1	64
DDR2	266	533	4,1	64
DDR2	333	667	5,2	64
DDR3	533	1066	8,3	64
DDR3	667	1333	10,4	64
DDR3	800	1600	12,5	64

Fonte: Intel (INTEL, 2011a)

As memórias do tipo DDR são amplamente usadas nos sistemas *Desktop*, *Laptops* e servidores, sendo encontradas também em sistemas embarcados com maior capacidade de processamento, como roteadores e *switches* de rede. A característica principal destas memórias é a capacidade de transmitir dados na borda de subida e na borda de descida do sinal de *clock* do barramento. Ao longo dos anos diversas versões foram lançadas, com diferentes capacidades de armazenamento e taxa de transferência de dados.

A tabela 2 mostra algumas das versões mais conhecidas dessas memórias. Na tabela 3 é indicado o tipo de memória DDR suportado para dois modelos de processadores. No caso do processador T7250, da família Core 2, o suporte é dependente do controlador (G)MCH utilizado e para o processador da família Core i7, o suporte depende apenas do processador já que o controlador de memória é integrado.

A banda teórica máxima da interface FSB 800 MHz, presente no processador T7250 indicado na tabela 3 pode ser calculada pela equação (1) :

$$banda_fsb = qtd_palavras \, x \, freq_barramento \, x \, tamanho_palavra \tag{1}$$

Neste caso, o barramento FSB é do tipo QDR, o que significa que a cada ciclo de *clock* são transferidas quatro palavras de dados ($qtd_palavras = 4$), a frequência do

barramento utilizado é de 200 MHz ($freq_barramento = 200Mhz$) e a palavra de dados é de 64 bits ($tamanho_palavra = 64bits = 8bytes$), logo $bw_fsb_{800Mhz} = 4\,x\,200MHz\,x\,8bytes = 6,4GB/s$.

Para aumentar a taxa de transferência no acesso às memórias, muitos sistemas suportam 2 canais independentes para os módulos de memória, o que permite duplicar a taxa de transferência máxima quando ambos os canais são utilizados simultaneamente (INTEL, 2011a).

Tabela 3: Características de alguns processadores

Proc.	Família	Ano	CPU [GHz]	Qtd. núcleos	MCH (Sugerido)	Memória	FSB [MHz]	DMI [GB/s]
T7250	Core 2	2007	2	2	Intel GME965	DDR2 533/667	800	1
i7-840QM	Core i7	2010	1,86	4	Integrado	DDR3 1066/1333	N.A.	2

Fonte: Manuais Intel (INTEL, 2011a)

Os diagramas em blocos das famílias Core 2 e Core i7 são apresentados nas figuras 1 e 2 respectivamente. A principal diferença é que na família Core i7, o controlador de memória MCH foi integrado ao processador, permitindo assim, um aumento expressivo na taxa de transferência de dados entre a CPU e a memória RAM. O componente PCH da família Core i7 faz o papel do componente ICH, gerenciando a comunicação entre a CPU e os dispositivos de I/O mais lentos, como por exemplo os discos rígidos e dispositivos USB.

Os componentes MCH e ICH também são conhecidos como *Northbridge* e *Southbridge* respectivamente.

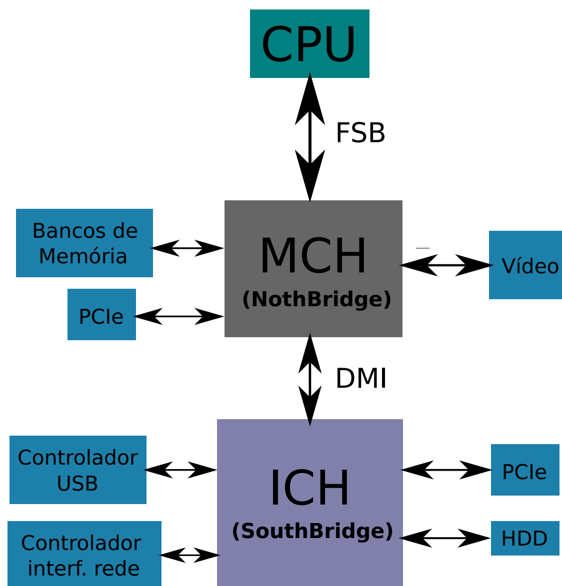


Figura 1: Diagrama em blocos do Intel *Core 2 Duo*.
Fonte: (INTEL, 2011a)

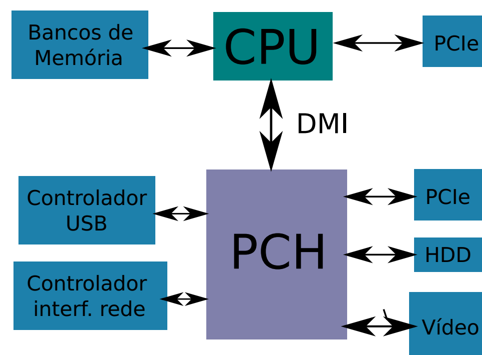


Figura 2: Diagrama em blocos do Intel *Core i7*.
Fonte: (INTEL, 2011a)

2.1.2 *PIPELINE* DO PROCESSADOR E *BRANCH-PREDICTION*

A execução de uma instrução pelo processador não é feita de forma instantânea, sendo composta de vários estágios sequenciais. Considerando uma operação envolvendo operandos em memória, uma sequência de 5 estágios poderia ser descrita por:

E1. busca da instrução na memória e incremento do ponteiro de instruções;

E2. decodificação da instrução para determinar o que executar;

E3. busca dos operandos;

E4. execução da operação;

E5. armazenamento do resultado na memória;

Para aproveitar de forma mais eficiente os ciclos (*clock*) do processador, normalmente são implementados *pipelines* que permitem que o processador inicie o tratamento da próxima instrução antes que a execução da instrução atual tenha sido finalizada. Desse modo, enquanto a instrução atual está no estágio 2 da execução, a instrução seguinte pode estar sendo executada no estágio 1 (TANENBAUM, 2007; PATTERSON; HENNESSY, 2008). A quantidade de estágios depende de cada processador e em alguns processadores modernos o *pipeline* pode ser composto de 10 ou mais estágios (TANENBAUM, 2007).

Etapa \ Tempo	T1	T2	T3	T4	T5	T6	T7	T8	T9
E1	A1	B1	C1	D1	E1				
E2		A2	B2	C2	D2	E2			
E3			A3	B3	C3	D3	E3		
E4				A4	B4	C4	D4	E4	
E5					A5	B5	C5	D5	E5

Figura 3: Execução de instruções no *pipeline*
Fonte: Autoria própria

A figura 3 exemplifica a execução de cinco instruções no *pipeline* de cinco estágios descrito anteriormente. Neste caso, foram necessários 9 períodos de tempo (ou ciclos do processador) para a execução total das 5 instruções. Sem a utilização do mecanismo de *pipeline*, conforme ilustrado na figura 4, são necessários 25 períodos de tempo para a execução de todas as instruções. Em ambos os casos, considerou-se que cada estágio é executado em 1 ciclo apenas.

Tempo	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Etapa	A1	A2	A3	A4	A5	B1	B2	B3	B4	B5
Tempo	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20
Etapa	C1	C2	C3	C4	C5	D1	D2	D3	D4	D5
Tempo	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30
Etapa	E1	E2	E3	E4	E5					

Figura 4: Execução de instruções sem *pipeline*
Fonte: Autoria própria

Todo o paralelismo na execução das instruções, obtido com o uso de *pipelines*, funciona de maneira mais eficiente em sequências de código linear, onde não são encontradas

execuções condicionais que podem desviar o fluxo de execução da aplicação. De maneira geral, as aplicações não apresentam um código linear e contêm uma grande quantidade de desvios condicionais.

Nos casos de desvio condicional, antes de iniciar a execução da instrução no *pipeline*, na maioria das situações é necessário fazer uma predição sobre qual "caminho" será executado pela aplicação para decidir qual instrução do desvio condicional deve ser carregada no *pipeline*. Caso a predição esteja incorreta, perde-se a oportunidade de execução no *pipeline*. Por esse motivo, alguns processadores utilizam um mecanismo, chamado de *branch prediction* (BP), para determinar qual é a melhor instrução a ser carregada, melhorando assim a utilização do mesmo. Se após o inicio do processamento da instrução selecionada detectar-se que a predição está incorreta, essa instrução precisa ser removida do *pipeline* ao custo de vários ciclos do processador. Neste momento, é contabilizado um evento de *branch-miss* pelo módulo de monitoramento do processador (INTEL, 2011b, Volume 3).

Na arquitetura superescalar, várias instruções podem ser executadas simultaneamente em um mesmo estágio do processamento, diferentemente do que ocorre no processamento paralelo executado pelo *pipeline*, no qual várias instruções são executadas em paralelo, porém todas estão em diferentes estágios da execução. Um processador com arquitetura superescalar não tem necessariamente vários *pipelines* (normalmente possui um *pipeline* apenas) e apenas alguns dos estágios do *pipeline*, geralmente os mais lentos, são expandidos para que múltiplas execuções possam ocorrer (TANENBAUM, 2007).

2.1.3 BARRAMENTO PCI EXPRESS

O barramento PCI Express tem sido utilizado para interconectar os dispositivos que necessitam uma conexão rápida com o processador. Este novo barramento é o sucessor do barramento PCI e suas variações como o AGP, PCI-X. Esses barramentos eram barramentos paralelos e compartilhados por diversos dispositivos do sistema, em contraste com o barramento serial ponto-a-ponto de alta-velocidade especificado para o PCI Express (BREWER; SEKEL, 2004).

A substituição do barramento paralelo por conexões seriais reflete uma tendência da indústria e permite uma maior escalabilidade, além de taxas mais altas de transferência de dados. O uso de conexões seriais permite eliminar problemas comuns das conexões paralelas, entre eles o atraso de propagação dos sinais, a interferência entre as linhas de sinal, a atenuação e a reflexão do sinal (HEIL, 2004). Essas características das conexões

seriais facilitam também o projeto das placas de circuito impresso (LEE, 2004). A escalabilidade é possível com a adição de *lanes* ao barramento, sem a necessidade de alterar a taxa de transmissão de dados ou preocupar-se com os problemas de propagação de sinais paralelos.

Uma característica importante dos barramentos PCI é que eles são do tipo *half-duplex*. Isto quer dizer que os dados trafegam em apenas um dos sentidos a cada instante de tempo. Em um dado instante eles seguem da CPU para o dispositivo e em outro do dispositivo para a CPU, limitando desse modo a capacidade máxima de transferência dos dispositivos (BREWER; SEKEL, 2004).

As taxas de transferência máximas para os barramentos PCI e algumas de suas variantes são apresentadas na tabela 4 e referem-se a capacidade de transferência em apenas um dos sentidos do barramento, sendo que os valores indicados são válidos para os barramentos com largura de 32 e 64 bits.

Com as taxas de transferência apresentadas, uma placa de rede Gigabit Ethernet saturaria um barramento PCI de 32 bits operando a 33 MHz pois esse dispositivo requer uma banda de 125 MB/s (1 Gbps) para operar em modo *half-duplex*. Normalmente, as placas de rede em uma LAN operam em modo *full-duplex*, exigindo com isso uma banda máxima de 2 Gbps.

O saturamento do barramento PCI satura também a conexão entre a *Southbridge* e a *Northbridge*, conforme indicado na figura 1. Com isso, dispositivos conectados às interfaces USB e de comunicação com os discos rígidos teriam a performance prejudicada (BREWER; SEKEL, 2004).

As arquiteturas PCI, PCI-X e PCIe são especificadas pelo PCI-SIG (PCI-SIG, 2011b). A conexão básica do PCIe utiliza-se de dois canais seriais, sendo um canal responsável pela recepção e outro pela transmissão dos dados, permitindo desse modo a transmissão e recepção simultânea dos dados. Cada canal é composto de dois sinais elétricos diferenciais de baixa-voltagem (LVDS), que tem como principais características o baixo consumo de energia e maior imunidade a ruído (BREWER; SEKEL, 2004). Os pares de transmissão e recepção juntos são chamados de *lane* e formam o componente básico de uma conexão PCIe.

O aumento na capacidade de transmissão de uma conexão PCIe é feito com a adição de *lanes* e a especificação indica as larguras de barramento x1, x4, x8, x16 e x32 (BREWER; SEKEL, 2004), onde x1 representa uma conexão com 1 *lane*, x4 é composto

Tabela 4: Largura de banda dos barramentos PCI e PCI-X

Barramento	Frequência [MHz]	Taxa de transferência de pico [1] [MB/s]	
		32 bits	64 bits
PCI	33	133	266
PCI	66	266	533
PCI-X	66	266	533
PCI-X	100[2]	400	800
PCI-X	133	533	1066
PCI-X	266	1066	2133
PCI-X	533	2133	4266

Fonte: (SOLOMON, 2008)

Notas:
[1] *Half-duplex*
[2] Variante do PCI-X 133

de 4 *lanes* e assim sucessivamente. Desse modo, a escalabilidade da interface PCIe pode ser garantida com mais facilidade. Com essa abordagem, os canais seriais são agrupados em conjuntos de canais paralelos mas que operam de forma independente.

As taxas básicas de transferência de dados para as versões 1.x, 2.x e 3.x da especificação PCIe são apresentadas na tabela 5. Essa taxa é contabilizada em número de transferências por segundo (GT/s), que para este caso é equivalente a Gbps (Giga bits por segundo). A codificação utilizada nas versões 1.x e 2.x é a 8b/10b, que codifica 8 *bits* de dados em uma palavra de 10 *bits*. Para a versão 3.x a codificação utilizada é a 128b/130b, considerada mais eficiente e que permite à versão 3.x dobrar a taxa de transferência de dados úteis (*payload*) (PCI-SIG, 2011a). É importante observar que a taxa de transferência básica da versão 3.x não dobra em relação à taxa básica para a versão 2.x, mas devido à eficiência da codificação 128b/130b, é possível dobrar a taxa de transferência de dados úteis.

As possíveis taxas de transferência para a versão 1.x são apresentadas na tabela 6. A taxa de dados codificados é obtida multiplicando-se a soma da taxa de transferência básica para o canal de TX e RX pela quantidade de *lanes* da implementação. Para a implementação x2 por exemplo, o resultado é : $dados_codificados = 2 * (2,5 + 2,5) = 5GT/s = 5Gbps$. Para se obter a taxa de dados decodificados (B), em Gbps, a partir da taxa de dados codificados (A), utiliza-se uma regra de três simples dada por: $B = A\ x\ taxa_codigo$. Para as versões 1.x e 2.x, o cálculo seria: $B = A\ x\ (8b/10b) = A\ x\ 0,8$.

Tabela 5: Taxa de transmissão básica por canal para o PCIe

Versão	Taxa de transf. básica [GT/s]	Codificação	Taxa de transf.[1] [MB/s]	Taxa de transf. (barramento x16)[2] [GB/s]
1.x	2,5	8b/10b	250	8
2.x	5		500	16
3.x	8	128b/130b	1000	32

Fonte: Adaptado de PCI-SIG FAQ (PCI-SIG, 2011a)

Notas:

[1] Valor aproximado para a taxa de transferência por canal/direção do *lane* (TX ou RX)

[2] Valor aproximado para a taxa de transferência total (TX+RX)

Tabela 6: Taxas de transferência para a interface PCIe versão 1.x

Implementação	Taxa de Dados		
	Codificados (8b/10b)	Decodificados	
	[Gbps]	[Gbps]	[GB/s]
x1	5	4	0,5
x4	20	16	2
x8	40	32	4
x16	80	64	8
x32	160	128	16

Fonte: Adaptado de Dell (BREWER; SEKEL, 2004)

2.1.4 GPU

As GPUs são unidades de processamento gráfico que compõem a placa de vídeo de muitos dos computadores de uso geral atuais e foram desenvolvidas inicialmente para retirar do processador principal (CPU) o processamento de elementos gráficos. Devido à sua grande capacidade de processamento paralelo, elas estão sendo utilizadas em diversas aplicações que requerem um grande poder computacional.

A arquitetura de *hardware* desses processadores não é a mesma da arquitetura x86 utilizada nas CPUs e por esse motivo, o binário gerado para as aplicações, precisa ser compilado especialmente para a GPU. Além disso, é necessário adaptar o código a ser executado na GPU, de modo a aproveitar ao máximo o paralelismo oferecido pela arquitetura. Os principais fabricantes de GPU disponibilizam ferramentas de desenvolvimento entre as quais estão a API CUDA, desenvolvida pela NVidia e o AMD APP SDK com suporte a OpenCL, oferecido pela AMD.

Abaixo, são apresentados alguns conceitos importantes adotados na arquitetura CUDA (PATTERSON; HENNESSY, 2008):

* *kernel* - programa ou função que será executada por várias *threads* na GPU;
* *bloco de threads* - conjunto de *threads* concorrentes que executam a mesma função ou trecho de código e podem se comunicar a fim de obter o resultado da execução;
* *grid* - conjunto de blocos de *threads* que executam o mesmo *kernel*.

A execução de um aplicativo na GPU, carregado a partir de uma CPU de uso geral, tem as seguintes etapas principais (PHARR; FERNANDO, 2005, Chapter 30):

* Alocação da memória necessária na GPU, a partir do *Host* (CPU);
* Cópia dos dados e instruções do *Host* para a GPU;
* Execução das *threads* na GPU;
* Cópia dos dados da GPU para o *Host*;
* Liberação da memória na GPU.

Um diagrama geral de uma placa GPU é mostrado na figura 5. A principal conexão da *GPU* é a interface *Host* - GPU, atualmente realizada através da interface

PCIe. Os primeiros modelos de *GPU* conectavam-se ao *Host* através de interfaces *AGP*, muito mais lentas que as interfaces *PCIe*, conforme apresentado na subseção 2.1.3.

Por se tratar de uma placa de vídeo, a GPU possui interface de saída de vídeo e um banco de memória para uso interno da GPU. A largura do barramento de conexão dessas memórias varia conforme o modelo do dispositivos e alguns exemplos são apresentados na tabela 7. A largura do barramento é maior que a utilizada nas memórias DDR dos computadores de uso geral apresentadas na tabela 2, sendo essa uma das razões para as altas taxas de transferências alcançadas na comunicação entre a GPU e o banco de memória interno.

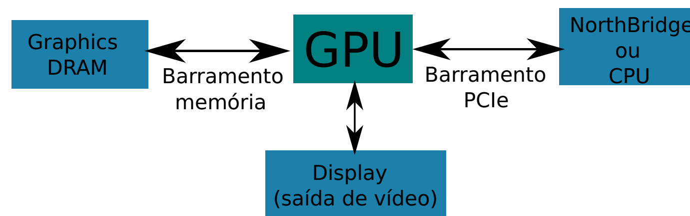


Figura 5: Diagrama geral da *GPU*
Fonte: Autoria própria.

Devido à grande variedade de modelos e configurações disponíveis para as GPUs, a NVidia disponibiliza uma planilha que permite determinar os melhores parâmetros a serem utilizados em uma determinada placa, melhorando assim a ocupação e uso da GPU. Essa planilha está disponível em (NVIDIA, 2011).

Tabela 7: Especificação de algumas placas GPU NVidia

Modelo	Especificação			
	Processador		Memória	
GeForce GTX 295 (2 GPUs)	Qtd. núcleos CUDA	2 x 240	*Clock* da interface	999 MHz
	Clock	1242 MHz	Configuração padrão	2x896 MB [1]
			Largura da interface	2x448 bits
			Largura de banda	223,8 GB/s
Quadro 6000	Qtd. núcleos CUDA	448		
	Single precision	1030,4 *Gigaflops*	Configuração padrão	6 GB [2]
	Double precision	515,2 *Gigaflops*	Largura da interface	384 bits
			Largura de banda	144 GB/s

Fonte: Adaptado de (NVI, 2011)

Notas:
[1] Memória do tipo GDDR3
[2] Memória do tipo GDDR5

2.2 O PADRÃO H.264

O padrão para codificação de vídeo H.264/AVC, também conhecido por MPEG-4 Part 10, foi desenvolvido pelo JVT, composto por integrantes dos grupos VCEG e MPEG, das organizações ITU-T e ISO/IEC respectivamente (WIEGAND et al., 2003c).

O padrão H.264 define a sintaxe e a semântica do vídeo codificado, tendo como foco o decodificador e as atividades envolvidas na decodificação (OSTERMANN et al., 2004). A estrutura básica do padrão H.264/AVC é similar à de padrões anteriores (H.261, MPEG-1, MPEG-2 / H.262, H.263 ou MPEG-4 part 2) (SULLIVAN et al., 2004), porém com uma maior eficiência na codificação, reduzindo o número de *bits* resultantes no vídeo codificado além de maior robustez a falhas na transmissão. Essas melhorias implicam em uma maior complexidade e consumo de recursos computacionais para o processo de codificação (WIEGAND et al., 2003c).

Para facilitar a adoção do H.264 nas mais diversas aplicações, o padrão define *profiles* e *levels*, que são conjuntos de capacidades e restrições que um decodificador H.264 pode suportar, com o objetivo de restringir a complexidade de cada equipamento ao mínimo necessário pois a implementação de todo o padrão pode não ser viável economicamente para um determinado tipo de aplicação. Desse modo, os *profiles* definem a sintaxe a ser utilizada na codificação, e consequentemente as ferramentas e recursos oferecidos pelo padrão e que devem ser utilizados na implementação. Os *levels* estão sempre vinculados a um determinado *profile* e definem restrições como por exemplo a quantidade de bits usadas na representação do vídeo ou então, a resolução máxima suportada (OSTERMANN et al., 2004). Inicialmente foram definidos os *profiles* Baseline, Main e Extended e posteriormente surgiram os *profiles* High e suas variantes.

Um diagrama em blocos simplificado de um codificador H.264 é apresentado na figura 6. Na sequência apresenta-se uma breve descrição de cada etapa representada pelos blocos.

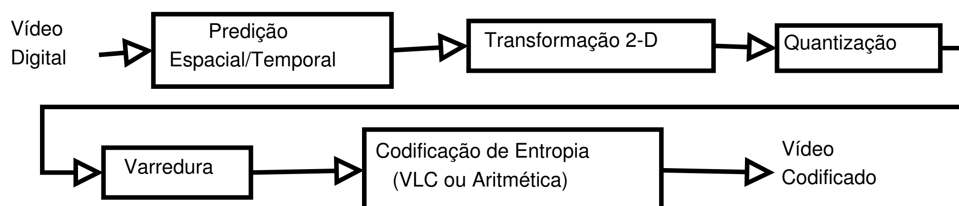


Figura 6: Diagrama em bloco da codificação H.264/*AVC*
Fonte: Adaptado de (SULLIVAN et al., 2004)

Cada quadro na entrada do codificador é dividido em macroblocos que é composto pelas componentes de luminância (Y) e de crominância (Cr e Cb). A quantidade de amostras de cada componente pode variar dependendo do *profile* ou de extensões do padrão(OSTERMANN et al., 2004).

Após a captura dos macroblocos na entrada do sistema, as amostras passam pela etapa de predição, que pode ser temporal ou espacial. Essas etapas são chamadas de predição *Inter-quadro* e *Intra-quadro* respectivamente. Na predição *Inter-quadro*, são consideradas as semelhanças entre macroblocos de quadros consecutivos de um fluxo de vídeo e a predição *Intra-quadro* realiza a predição de amostras do macrobloco com base em informações de macroblocos já transmitidos de um mesmo quadro, de forma análoga ao processo utilizado na codificação de imagens estáticas. A predição *Inter-quadro* utiliza-se de um mecanismo conhecido como compensação de movimento e gera um vetor de deslocamento que é transmitido com o vídeo codificado e posiciona o macrobloco atual em relação a um quadro de referência previamente codificado, contribuindo dessa forma para o aumento da taxa de compressão pois esse etapa reduz a quantidade de informação a ser codificada, a partir da eliminação de informações redundantes *Inter-quadros* e *Intra-quadros.*

Assim como nos padrões de codificação anteriores, o H.264 possui diversos modos usados para fazer a predição do macrobloco *Inter-quadro* e *Intra-quadro* e a seleção de cada modo usa um mecanismo chamado de RDO, que tem por função determinar o modo que permite atingir a melhor eficiência na codificação (WIEGAND et al., 2003a). Esse mecanismo pode ser considerado um dos fatores para o sucesso do H.264 em termos de taxa de compressão e qualidade visual (LIN et al., 2010). Para calcular o erro de predição, os algoritmos mais comuns usados pelos codificadores são a SAD e a SATD.

As equações (2) e (3) mostram o cálculo da SAD e SATD para um matriz do erro de predição $\boldsymbol{Diff}$, de tamanho 4x4, onde $\boldsymbol{Diff} = \boldsymbol{C} - \boldsymbol{A}$, $\boldsymbol{R}$ representa o macrobloco de referência e $\boldsymbol{A}$ corresponde ao macrobloco atual.

$$SAD(Diff) = \sum_{i=1}^{4}\sum_{j=1}^{4} |diff_{ij}| \tag{2}$$

$$SATD(DiffT) = (\sum_{i=1}^{4}\sum_{j=1}^{4} |diffT_{ij}|)/2 \tag{3}$$

A SAD é calculada diretamente sobre a matriz $\boldsymbol{Diff}$ e a SATD é calculada sobre a

$$H_{4x4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

Figura 7: Matriz de Hadamard
Fonte: Autoria própria

matriz $DiffT$ que foi obtida a partir de uma transformada 2-D de Hadamard sobre a matriz $Diff$, conforme indicado na equação (4). Os elementos $diff_{ij}$ e $diffT_{ij}$ representam o elemento da linha i e coluna j das matrizes $Diff$ e $DiffT$, respectivamente.

$$DiffT = H_{4x4}.Diff_{4x4}.H_{4x4}^{T} \tag{4}$$

Maiores informações sobre essas duas métricas podem ser obtidas em (ABDELAZIMA et al., 2010).

A etapa de transformação permite diminuir a redundância espacial das amostras (OSTERMANN et al., 2004) convertendo as amostras do sinal de vídeo do domínio espacial para o domínio da frequência, utilizando a Transformação de Inteiros, em substituição à DCT utilizada em padrões anteriores.

Na etapa de quantização, cada amostra é dividida pelo parâmetro de quantização (Qp) que pode ser diferente para cada macrobloco e que para amostras de 8-bits pode assumir 52 possíveis valores (SULLIVAN et al., 2004).

Após a transformação e quantização, a varredura é realizada sobre a matriz resultante das etapas anteriores com o objetivo de colocar os coeficientes de maior variância no início da sequência de coeficientes. A varredura também tem por objetivo, maximizar o número de coeficientes zerados consecutivos e assim representá-los de uma forma bastante compacta na etapa de codificação de entropia. O tipo de varredura também varia com o tipo de predição utilizada (SULLIVAN et al., 2004).

Um exemplo de varredura em zig-zag para um macrobloco de tamanho 4x4 é apresentado na figura 8. Considerando o macrobloco uma matriz T de tamanho 4x4, após a varredura, a sequência de coeficientes serializados para a codificação de entropia é $T_{zz} = t_{11}\ t_{12}\ t_{21}\ t_{31}\ t_{22}\ t_{13}\ t_{14}\ t_{23}\ t_{32}\ t_{41}\ t_{42}\ t_{33}\ t_{24}\ t_{34}\ t_{43}\ t_{44}$, sendo t_{ij} o elemento da linha i e coluna j da matriz T.

A codificação de entropia é uma etapa que não gera perda de informação do vídeo

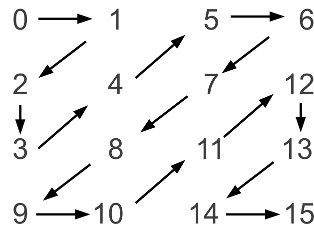


Figura 8: Varredura em zig-zag
Fonte: Autoria própria

e é responsável pela efetiva compactação dos dados. O objetivo é representar os dados de maior ocorrência por um código de comprimento menor, com o uso de códigos de tamanho variável (VLCs). Diferentemente dos padrões anteriores, que utilizam códigos VLCs fixos, no H.264 utiliza-se a CAVLC e a CABAC, que são codificações adaptativas ao contexto (OSTERMANN et al., 2004).

O desempenho do padrão H.264 em relação aos padrões anteriores pode ser visto na tabela 8. Para um cenário de *video streaming* com a utilização do *Main Profile* (MP) no codificador H.264, foi possível atingir uma redução de até 63,57 % em relação ao codificador MPEG-2. Em um cenário de vídeo conferência, utilizando o *Baseline Profile* (BP), a redução foi de até 40,49 % quando comparado com o codificador H.263 Base.

Tabela 8: Redução média na taxa de bits para o H.264

	Codificador	MPEG-4 ASP	H.263 HLP	MPEG-2
Video Streaming	Redução [%] (H.264/AVC MP)	37,44	47,58	63,57
	Codificador	H.263 CHC	MPEG-4 SP	H.263 Base
Vídeo conferência	Redução [%] (H.264/AVC BP)	27,69	29,37	40,59

Fonte: Adaptado de (WIEGAND et al., 2003b)

2.3 TÉCNICAS PARA MELHORAR A PERFORMANCE DO *SOFTWARE*

Diversas técnicas foram desenvolvidas ao longo dos anos para melhorar a performance de *software*. Entre as técnicas disponíveis, esta seção apresenta algumas que são mais comumente utilizadas em aplicações de processamento de imagens e multimídia.

Algumas das técnicas apresentadas requerem um conhecimento prévio da arquitetura de *hardware* a ser utilizada pois estão intrinsecamente ligadas a ela, como é o caso das instruções SIMD e a linguagem *assembly*.

Para o caso dos padrões de codificação de vídeo, como o H.264, essas técnicas são frequentemente utilizadas nas implementações dos codificadores.

2.3.1 USO DE INSTRUÇÕES SIMD

As instruções SIMD como MMX e SSE foram especialmente desenvolvidas para aplicações multimídia e caracterizam-se por processarem múltiplos dados com a utilização de uma única instrução do processador. Várias arquiteturas de *hardware* possuem seu conjunto de instruções SIMD sendo o seu uso bastante comum em DSPs. O conjunto de instruções SIMD é específico para cada arquitetura de *hardware* e mesmo dentro de uma arquitetura específica. Por isso, é importante mapear o subconjunto de instruções que é suportado para um dispositivo em especial.

Na metade da década de 1990, a Intel introduziu a extensão MMX na família de processadores Pentium e a extensão 3DNow foi introduzida na família K6-2 pela AMD. Depois disso, a extensão SSE foi criada e incrementada por ambas as companhias e hoje existem vários subconjuntos dessas instruções, como por exemplo SSE, SSE2, SSE3, SSE4. Por esse motivo, o desenvolvedor precisa conhecer o conjunto de instruções suportado pelo processador a ser utilizado antes de iniciar a implementação.

Um exemplo do uso de instruções SIMD para a execução paralela de diversas operações de soma sobre operandos de 8 bits é exibido na figura 9. Com o uso da rotina **__mm__add_epi8** são executadas 16 operações simultâneas sobre os vetores de dados. Esses vetores foram declarados com o tipo **__m128i** e apontam para os dados declarados como **char**.

```
1   /* soma de y e z com o resultado armazenado em x */
2
3   #define TAM 256
4   char x[ TAM ], y[ TAM ], z[ TAM ];
5   __m128i *vetx, *vety, *vetz;  /* tipo de dado especial */
6
7   /* inicializa os ponteiros */
8   vetx = (__m128i*)x;
9   vety = (__m128i*)y;
10  vetz = (__m128i*)z;
11
12  for (i = 0; i < TAM; i++) {
13          x[i] = y[i] + z[i];
14  }
15
16  for (i = 0; i < TAM/16; i++) {
17          vetx[i] = _mm_add_epi8(vety[i], vetz[i]);
18  }
```

Figura 9: Exemplo do uso de instruções SSE (SIMD)
Fonte: Autoria própria

Todas as instruções SIMD são um subconjunto das instruções de máquina de cada arquitetura de *hardware* e podem ser utilizadas de diversas maneiras na implementação de

software. É possível usar na forma *inline*, com o uso de bibliotecas especializadas ou então com a utilização de macros e tipos de dados dependentes de cada compilador (TROCKI, 2008).

2.3.2 APLICAÇÃO DE AUTO-VETORIZAÇÃO

Vetorização é uma técnica desenvolvida inicialmente para processadores vetoriais, que tem como característica principal a execução de diversas instruções em paralelo mesmo em sistemas com um único núcleo de processamento (*single core*). Esses processadores foram introduzidos no mercado na década de 1970. Para utilizar de forma eficiente o *hardware* de processamento vetorial, as aplicações eram reescritas de forma que as operações a serem executadas sobre um vetor de dados, fossem executadas simultaneamente sobre vários elementos do vetor ao invés de serem executadas de forma serializada, sobre um elemento de cada vez. Esse mesmo principio pode ser aplicado nos dias de hoje com a utilização de instruções SIMD (NAISHLOS, 2004), tanto em processadores de uso geral como nos processadores especializados em processamento de sinais (DSPs).

A auto-vetorização consiste na aplicação automática da vetorização por parte do compilador (NAISHLOS, 2004). Com isso, os laços de execução são reescritos automaticamente para a forma vetorial, permitindo o uso das instruções SIMD para a melhora da performance da aplicação. O compilador GCC (GCC, 2011b) possui suporte a auto-vetorização, que pode ser habilitada com o uso da *flag -ftree-vectorize* ou com o uso da *flag -O3* que por padrão habilita a auto-vetorização (GCC, 2011a).

2.3.3 BIBLIOTECAS ESPECIALIZADAS

Muitas empresas, como as fabricantes de DSPs e de compiladores, oferecem bibliotecas com funções e classes otimizadas, algumas delas fazendo uso intensivo de instruções SIMD para o processamento dos dados multimídia. Na maioria dos casos, essas bibliotecas apresentam uma performance melhor quando comparadas com as bibliotecas tradicionais, porém o *software* precisa ser implementado e/ou adaptado para utilizar a API da biblioteca selecionada.

Uma variedade de bibliotecas para as mais diversas aplicações multimídia e em diferentes formatos de licenciamento estão disponíveis, entre elas pode-se citar a Intel IPP (INTEL, 2011c), a VA-API (FREEDESKTOP, 2011) e a OpenMax (KHRONOS, 2011).

2.3.4 USO DE LINGUAGEM ASSEMBLY

É possível integrar trechos de código em linguagem de baixo nível com trechos em linguagem de mais alto nível com a utilização do *assembly Inline*, normalmente com o uso de palavras-chave como por exemplo *_asm*. A sintaxe pode variar entre os diversos compiladores mas permite uma fácil substituição de trechos críticos do *software* por instruções mais eficientes e compactas, com pequeno impacto na estrutura do *software* (COPPEN, 2005).

Em casos específicos, uma boa performance pode ser alcançada com o uso desta técnica, ao custo de portabilidade muito baixa para o *software*. A principal razão são os diferentes conjuntos de instruções de cada arquitetura de *hardware* e também da diferença na sintaxe da linguagem *assembly* entre os compiladores. A tabela 9 apresenta a diferença na sintaxe para algumas operações que podem ser executadas na arquitetura x86. Nesse caso, são apresentadas as formas conhecidas como sintaxe AT&T e sintaxe Intel.

Não existe uma padronização para a sintaxe *assembly* na arquitetura x86 e caso um *software* precise ser compilado com um montador (*assembler*) que não suporte a sintaxe usada na implementação original, o *software* terá que ser adaptado.

Tabela 9: Diferenças na sintaxe *assembly* para a arquitetura x86

Código Intel		Código AT&T	
mov	eax,1	movl	$1,%eax
mov	ebx,0ffh	movl	$0xff,%ebx
int	80h	int	$0x80
mov	ebx, eax	movl	%eax, %ebx
mov	eax,[ecx]	movl	(%ecx),%eax
mov	eax,[ebx+3]	movl	3(%ebx),%eax
mov	eax,[ebx+20h]	movl	0x20(%ebx),%eax
add	eax,[ebx+ecx*2h]	addl	(%ebx,%ecx,0x2),%eax
lea	eax,[ebx+ecx]	leal	(%ebx,%ecx),%eax
sub	eax,[ebx+ecx*4h-20h]	subl	-0x20(%ebx,%ecx,0x4),%eax

Fonte: (SANDEEP, 2003)

2.4 MELHORIA DA PERFORMANCE DE CODIFICADORES DE VÍDEO

Alguns dos trabalhos apresentados a seguir estão relacionados à otimização de codificadores de padrões anteriores ao H.264, como por exemplo o MPEG-4 Part 2 e o H.263. Mesmo assim, eles são uma boa fonte de informação, principalmente quando abordam questões relacionadas a algoritmos do codificador. Muitos dos mecanismos utilizados

em todos os padrões de codificação citados são equivalentes, muitas vezes mudando apenas o algoritmo ou a opção de algoritmos e modos de operação disponíveis.

2.4.1 MELHORIA NO ACESSO À MEMÓRIA

Em (NASIM S. MASUD et al., 2005), o objeto de estudo foi um codificador MPEG-4 Part 2, onde o objetivo principal foi a minimização do acesso à memória e cópia de dados. Levou-se em consideração o fato que a velocidade da CPU é muito superior à velocidade de acesso ao barramentos e memórias externas dos sistemas microprocessados atuais. Em resumo, é apresentada uma abordagem para melhorar a performance do codificador reduzindo-se a quantidade de acessos à memória do sistema.

Citou-se que a estimação de movimento (ME) consome de 40 a 60% da carga da CPU, mas nenhuma mudança no algoritmo de ME foi proposta. No codificador MPEG-4 Part 2 de referência, o ME é aplicado a todos os quadros e as diferenças de todo o quadro são armazenadas em um *buffer*.

No modelo proposto, apenas o vetor de movimento (MV) e as informações de modo são armazenados e não mais as informações de diferença, melhorando assim a utilização da memória. A ME em sub-pixel (ex. 1/4 de pixel), requer interpolação e as informações sobre a interpolação são armazenadas em um *buffer* que é muitas vezes maior que o tamanho de um quadro. Para a interpolação, a memória é utilizada de forma ineficiente pois apenas uma pequena fração da informação é usada e a ME é aplicada a cada macrobloco. É suficiente interpolar apenas o macrobloco que está sendo processado ao invés de interpolar o quadro inteiro. Outra melhoria possível é utilizar toda a largura do barramento nas transferências de memória, isto é, em um barramento de 32 bits, faz-se a leitura com um ponteiro para 32 bits ao invés de 4 leituras de 8 bits. O resultado obtido foi um codificador 15 a 50 vezes mais rápido que a versão sem otimização.

2.4.2 MELHORIA NO ACESSO À MEMÓRIA E USO DA LINGUAGEM *ASSEMBLY*

Um codificador H.263 foi desenvolvido em (SHEIKH et al., 2008) com base em uma versão do codificador implementado na Universidade British Columbia (UBC) e o codificador resultante ficou 61 vezes mais rápido e foi executado em um DSP da Texas Instruments. Verificou-se que uma parcela significativa do tempo gasto na codificação de vídeo é gasto no acesso a grande quantidade de dados, normalmente disponível no barramento de memória externa do processador. Para a versão não otimizada do codificador,

observou-se que o cálculo da SAD consumia 67% do tempo de codificação. Foram apresentados também o número de ciclos de operação necessários para as principais funções do codificador, antes e depois da otimização. A solução apresentada é específica para o *hardware* utilizado e foi obtida com o uso eficiente da memória *on-chip* para dados e código, além da reescrita de partes do código para a linguagem *assembly*. Com a melhor utilização da memória interna (*on-chip*), obteve-se um aumento de 29 vezes na velocidade de codificação. A melhoria do código resultou em um aumento de 4 vezes na velocidade de codificação e quando as duas abordagens foram aplicadas simultaneamente, o resultado final foi um aumento de 61 vezes na velocidade de codificação. Os resultados obtidos podem ser vistos na tabela 10.

Tabela 10: Número de ciclos (x1000) para as principais rotinas no H.263

Operação	Código Original	Melhorias		
		Memória	Código	Memória e Código
Motion Estimation	1356000	33400	325000	13000
Motion Compensation	26200	4200	6400	3400
DCT	17200	670	6000	130
Quantization	7700	660	3300	660
Interpolation	16900	4200	2200	750
Reconstruction	31000	4000	9100	2260
Outros	52000	8300	31200	6270
Total	1476000	51400	374000	24200

Fonte: (SHEIKH et al., 2008)

2.4.3 MELHORIAS NA MANIPULAÇÃO DE DADOS

Um projeto centrado na melhor utilização da memória para um codificador de vídeo MPEG-4 Part 2 foi proposto em (DENOLF et al., 2005). Levando-se em consideração que os algoritmos de codificação de vídeo têm como foco principal a manipulação de dados, a transferência e armazenamento desses dados é o fator principal de custo na eficiência do *software*. Neste contexto, foi sugerida uma sequência de passos a serem adotados no projeto para sistemas embarcados. Os passos são os seguintes:

1. Análise do algoritmo, com o objetivo de identificar gargalos e possíveis candidatos a serem melhorados. Essa etapa pode ser desenvolvida com o auxílio de ferramentas de *software profiling*;

2. Ajuste do algoritmo, para simplificação das regiões não envolvidas com a norma/-padrão, como os algoritmos de *Rate Control* e ME, de modo a reduzir o seu custo de implementação e/ou melhorar o uso de memória;

3. Otimizações para o uso da memória, minimizando a frequência de acesso, evitando grandes acessos à memória e reutilizando dados para melhorar a localidade dos mesmos;

4. Seleção da arquitetura e implementação final, onde são definidas a arquitetura de *hardware* a ser utilizada, podendo-se para isso, fazer uso das informações de *profiling* obtidas nas etapas anteriores sobre o código otimizado.

Como resultado, a análise do perfil da aplicação com ferramentas automatizadas e os dados da simulação permitiram uma redução do código em aproximadamente 30% passando de um *software* com mais de 67 mil linhas de código, para 22 mil linhas e o resultado final otimizado ficou com 9 mil linhas de código. Nesta etapa, foram identificadas as tarefas que demandavam mais recurso computacional e verificou-se a dominância dos dados na aplicação. A tabela 11 apresenta o número de acessos à memória para cada uma das funções principais do codificador.

Tabela 11: Acesso à memória das funções principais do codificador original

Função	Frequência de acesso [Maccessos/s]	Num. de acessos relativos [%]	Num. de ciclos relativos [%]
Motion estimation	4149,7	83,9	84,2
Texture coding	286,6	5,8	5,8
Motion compensation	99,6	2,0	1,8
Capture	19,0	0,4	0,7
Reconstruct	26,5	0,5	0,3
Calculate error	11,4	0,2	0,2
Manipulate frame	113,0	2,3	1,5
Padding	53,6	1,1	1,5
Outros	186,3	3,8	4,1

Fonte: (DENOLF et al., 2005)

Os algoritmos de *Rate Control* e ME também foram melhorados e foi implementada a detecção antecipada dos coeficientes zerados após a quantização.

A alta correlação entre o número de acessos à memória e o número de ciclos consumidos pelo módulo reforçam a ideia de que o processo de codificação de vídeo é

fortemente ligado à manipulação de dados e as operações aritméticas por exemplo, tem um peso menor.

2.4.4 IMPLEMENTAÇÃO DE UM CODIFICADOR H.264 BASELINE EM DSP

A implementação de um codificador H.264 *baseline* para o DSP TMS320DM642 foi apresentada em (GUNEY, 2006). A versão inicial processava 3,1 fps chegando a 26,7 fps para a versão final. Dois algoritmos de ME foram implementados sendo o *full search* e o *three step hierachical search* e técnicas de otimização de *software* e compilação foram aplicadas sobre o codificador implementado.

O resultado da implementação usando-se apenas as bibliotecas fornecidas pela Texas Instruments, além do uso de opções de configuração do compilador e o particionamento adequado das memórias *on-chip* do DSP são apresentados na tabela 12. Os resultados obtidos com a versão final são apresentados na tabela 13. Nessa tabela, a melhoria no padrão de acesso à memória refere-se à readequação do código para redução da quantidade de acessos ao barramento externo de memória e os consequentes bloqueios da CPU devido à dinâmica do acesso a esse barramento. As otimizações de código e compilação sem as melhorias no padrão de acesso à memória permitiram ao codificador atingir uma velocidade de codificação de apenas 10 fps, como pode ser observado na tabela 12. Quando essas otimizações foram utilizadas em conjunto com a melhoria no padrão de acesso, foi possível atingir a velocidade de 24,25 fps, como indicado na tabela 13. Para a mudança no tipo de variáveis, a velocidade de codificação foi de 26,31 fps, obtida com a mudança no tipo de variáveis de *integer* (32 bits) para *short* (16 bits) em algumas das estruturas de dados. Essa mudança permitiu uma melhor utilização do barramento externo de 16 bits do DSP utilizado. A alocação dos dados mais acessados na memória interna do dispositivo, permitiu aumentar um pouco mais a velocidade pois essa memória interna tem uma velocidade de acesso maior mas apresenta uma menor capacidade de armazenamento e por esse motivo, é necessário fazer uma seleção criteriosa dos dados que serão armazenados nela.

Assim como em outras referências apresentadas, o acesso à memória foi um dos gargalos para o desempenho do codificador. A otimização apenas do *software* não permitiu um grande ganho de performance no codificador, ficando este limitado a uma velocidade em torno de 10 fps. Quando a otimização no acesso à memória e as otimizações de *software* e compilação foram utilizadas em conjunto, foi possível atingir a velocidade de 26,7 fps.

Tabela 12: Performance do codificador com a otimização do *software*

	Total de ciclos	Velocidade [fps]
Versão inicial (sem otimização)	$2,17x10^8$	3,31
Versão com otimizações de código e compilação	$6,92x10^7$	10,4

Fonte: (GUNEY, 2006)

Tabela 13: Número total de ciclos da CPU para as etapas de otimização

Etapa	Ciclos médios para 1 quadro	Veloc. [fps]
Código sem modificação	217.914.368	3,31
Particionamento L2 RAM/Cache	217.904.555	3,31
Melhoria no padrão de acesso à memória	194.314.544	3,71
Otimizações de código e compilação	29.679.631	24,25
Mudança no tipo das variáveis	27.365.153	26,31
Alocação de dados na memória interna	27.282.428	26,40
Velocidade média final para 20 quadros		26,70

Fonte: (GUNEY, 2006)

2.5 *SOFTWARE PROFILING*

A identificação dos pontos críticos para a execução de uma aplicação geralmente é feita com o auxílio de ferramentas que fazem a análise dinâmica, coletando as informações durante a execução da aplicação. Existem diversas ferramentas disponíveis, sendo que cada ferramenta possui uma diversidade de opções e características (GAMBLIN, 2009).

Nesta seção serão apresentadas algumas opções disponíveis.

2.5.1 GPROF

Com a ferramenta gprof (GPROF, 2011a), a aplicação sob análise precisa ser compilada com a *flag* -pg para o GCC. Com isso, um binário especial será produzido e após a execução deste, o arquivo gmon.out será automaticamente gerado. Este arquivo de saída deve ser processado pelo gprof para assim gerar os relatórios de execução da aplicação (GPROF, 2011b). O gprof é disponibilizado junto com o pacote GNU binutils (GNU, 2011) e fornece três opções de relatório: *flat profile*, *call graph* e *annotated source*. Um exemplo para cada um desses relatorios pode ser visto nas figuras 10, 11 e 12. A função **progress_update()** foi destacada nos relatórios e devido ao tamanho de cada um deles, alguns trechos foram suprimidos e substituídas por **(...)**.

O relatório *flat* apresenta o tempo total gasto pela aplicação, assim com a quantidade de vezes que cada função foi chamada. O *call graph* exibe o tempo gasto por cada função e as funções filhas, permitindo relacionar a dependência entre cada uma delas. O código fonte de uma função especifica pode ser visualizado com o relatório *annotated source*, caso o código fonte esteja acessível durante a execução do gprof. A quantidade de vezes que a função foi chamada também é indicada nesse relatório.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      0.01     0.01     1335     0.01     0.01  ptimer_measure
  0.00      0.01     0.00     4004     0.00     0.00  ptimer_read
  0.00      0.01     0.00     1337     0.00     0.00  poll_internal
  0.00      0.01     0.00     1337     0.00     0.00  posix_measure
  0.00      0.01     0.00     1336     0.00     0.00  select_fd
  0.00      0.01     0.00     1336     0.00     0.00  sock_poll
  0.00      0.01     0.00     1335     0.00     0.00  fd_read
  0.00      0.01     0.00     1335     0.00     0.00  posix_diff
  0.00      0.01     0.00     1335     0.00     0.00  sock_read
  0.00      0.01     0.00     1334     0.00     0.00  bar_update
  0.00      0.01     0.00     1334     0.00     0.00  progress_update
  0.00      0.01     0.00     1334     0.00     0.00  update_speed_ring
  0.00      0.01     0.00     1334     0.00     0.00  write_data
  0.00      0.01     0.00      340     0.00     0.00  get_log_fp
  0.00      0.01     0.00      171     0.00     0.00  check_redirect_output
  0.00      0.01     0.00      170     0.00     0.00  logflush
  0.00      0.01     0.00      160     0.00     0.00  logputs
  0.00      0.01     0.00      154     0.00     0.00  log_set_save_context
  0.00      0.01     0.00       79     0.00     0.00  count_cols
  0.00      0.01     0.00       77     0.00     0.00  calc_rate
  0.00      0.01     0.00       77     0.00     0.00  create_image
  0.00      0.01     0.00       77     0.00     0.00  display_image
  0.00      0.01     0.00       77     0.00     0.00  get_grouping_data
  0.00      0.01     0.00       77     0.00     0.00  with_thousand_seps
  0.00      0.01     0.00       62     0.00     0.00  eta_to_human_short
   (...)
```

Figura 10: Exemplo de execução do *gprof flat report*
Fonte: Autoria própria

Durante a execução da aplicação, cada chamada de função executa também a função **mcount**, de modo a coletar em memória as informações sobre quem chamou a função atual e quantas vezes ela foi chamada. Quando a aplicação é finalizada, as informações são gravadas no arquivo gmon.out (GPROF, 2011b). Esse processo de coleta de dados acarreta em uma execução mais lenta da aplicação em análise.

Para as funções da biblioteca C, uma versão compilada com a opção -pg é utilizada para que todas as funções utilizadas façam a coleta das informações de *profiling* (GPROF, 2011b).

2.5.2 *HARDWARE PERFORMANCE COUNTERS*

A maioria das famílias de processadores atuais possuem recursos que permitem o monitoramento da performance do sistema. Esses recursos caracterizam-se por um conjunto de registradores que contabilizam eventos que ocorrem em determinado módulo ou interface do processador. Esses eventos podem ser o número de ciclos de *clock* totais

```
                       Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 100.00% of 0.01 seconds

index % time    self  children    called     name
                0.00    0.00    1/1335           main [2]
                0.01    0.00 1334/1335           fd_read_body [3]
[1]    100.0    0.01    0.00 1335             ptimer_measure [1]
                0.00    0.00 1335/1337           posix_measure [9]
                0.00    0.00 1335/1335           posix_diff [13]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    0.01                 main [2]
                0.00    0.01    1/1              retrieve_url [6]
                0.00    0.00    1/1335           ptimer_measure [1]
                0.00    0.00    2/2              rpl_getopt_long [88]
                0.00    0.00    1/2              ptimer_new [83]
                0.00    0.00    1/1              i18n_initialize [141]
                0.00    0.00    1/1              defaults [120]
                0.00    0.00    1/1              init_switches [144]
                0.00    0.00    1/1              initialize [145]
                0.00    0.00    1/1              find_locale [128]
                0.00    0.00    1/1
set_progress_implementation [182]
                0.00    0.00    1/1              rewrite_shorthand_url [178]
                0.00    0.00    1/18             xstrdup [40]
                0.00    0.00    1/1              log_init [152]
                0.00    0.00    1/1              iri_new [147]
                0.00    0.00    1/1              set_uri_encoding [183]
                0.00    0.00    1/1              url_parse [193]
                0.00    0.00    1/1              url_free [191]
                0.00    0.00    1/1              iri_free [146]
                0.00    0.00    1/1              log_close [151]
                0.00    0.00    1/1              cleanup [112]
                0.00    0.00    1/1              get_exit_status [130]
-----------------------------------------------
                0.00    0.01    1/1              gethttp [4]
[3]     99.9    0.00    0.01    1            fd_read_body [3]
                0.01    0.00 1334/1335           ptimer_measure [1]
                0.00    0.00 4004/4004           ptimer_read [7]
                0.00    0.00 1334/1335           fd_read [12]
                0.00    0.00 1334/1334           write_data [18]
                0.00    0.00 1334/1334           progress_update [16]
                0.00    0.00    1/46             xmalloc [34]
                0.00    0.00    1/1              progress_create [160]
                0.00    0.00    1/1              progress_interactive_p [162]
                0.00    0.00    1/2              ptimer_new [83]
                0.00    0.00    1/1              progress_finish [161]
                0.00    0.00    1/1              ptimer_destroy [163]
-----------------------------------------------
                0.00    0.01    1/1              http_loop [5]
[4]     99.9    0.00    0.01    1            gethttp [4]
(...)
```

Figura 11: Exemplo de execução do *gprof call graph report*
Fonte: Autoria própria

```
*** File wget-1.13.4/src/progress.c:
(...)
                void
                progress_update (void *progress, wgint howmuch, double dltime)
        1334 -> {
                  current_impl->update (progress, howmuch, dltime);
                }

                /* Tell the progress gauge to clean up.
Calling this will free the
                   PROGRESS object, the further use of which is not allowed.  */

                void
                progress_finish (void *progress, double dltime)
                {
                  current_impl->finish (progress, dltime);
                }


                /* Dot-printing. */

                struct dot_progress {
                  wgint initial_length;
/* how many bytes have been downloaded
                                                   previously. */
                  wgint total_length;
/* expected total byte count when the
                                                   download finishes */

                  int accumulated;
/* number of bytes accumulated after
                                                   the last printed dot */

                  int rows;
/* number of rows printed so far */
                  int dots;
/* number of dots printed in this row */

                  double last_timer_value;
                };

(...)

Top 10 Lines:
     Line       Count
      183        1334

Execution Summary:
        1    Executable lines in this file
        1    Lines executed
   100.00    Percent of the file executed

     1334    Total number of line executions
  1334.00    Average executions per line
```

Figura 12: Exemplo de execução do *gprof annotated source report*
Fonte: Autoria própria

do processador, número de ciclos no acesso ao barramento externo ou eventos diversos da memória *cache* entre outros (INTEL, 2011b, Volume 3 - Chapter 30). Ao atingir um determinado limiar, uma interrupção pode ser gerada e essas informações podem ser então utilizadas na análise da aplicação em execução (KERNEL.ORG, 2011; PAPI, 2011).

Nos processadores Intel, essa função é realizada pela PMU, responsável por gerenciar o monitoramento dos eventos e disponibilizá-los para o sistema operacional. Esses eventos geralmente são monitorados através de ferramentas como a VTune da Intel (INTEL, 2011a), a API PAPI (PAPI, 2011) ou o conjunto de ferramentas perf (KERNEL.ORG, 2011) disponíveis no sistema operacional Linux.

Com base em alguns desses contadores e nas equações sugeridas em (INTEL, 2009) é possível estimar o potencial de otimização do *software* atual e o impacto de alguns eventos sobre a performance do sistema equipado com processadores Intel. Neste trabalho são utilizadas as taxas CPI e BMPI, representadas pelas equações (5) e (6) respectivamente.

$$CPI = \frac{CPU_CLK_UNHALTED.CORE}{INST_RETIRED.ANY} \tag{5}$$

A taxa CPI representa a relação entre a quantidade de *clocks* do processador quando ele está operando efetivamente (não está no estado *halt*) e a quantidade de instruções que foram executadas por completo. O valor ideal para a arquitetura Intel Core 2 fica em torno de 0,25. Quanto maior o valor obtido para uma aplicação, maior é a oportunidade de otimização da mesma.

$$BMPI = \frac{RESOURCE_STALLS.BR_MISS_CLEAR}{CPU_CLK_UNHALTED.CORE} * 100 \tag{6}$$

O percentual de ciclos gastos para o processador se recuperar de um erro na predição de desvios (*branch prediction*) durante a operação do *pipeline* pode ser obtido com a taxa BMPI (INTEL, 2009). Com essa taxa, é possível identificar aplicações que, devido às suas características internas, não estão utilizando de forma eficiente a capacidade de execução em paralelo possibilitada com o uso do *pipeline* do processador.

Um exemplo de análise da saturação do barramento de memória através da utilização de contadores de performance em processadores Intel pode ser visto em (INTEL, 2010).

2.5.3 LINUX *PERF TOOLS*

O Linux oferece um conjunto de ferramentas para a análise da performance a partir dos recursos da PMU além de recursos próprios do sistema operacional como contadores de *software* e *tracepoints* para monitoramento de eventos e interfaces de *software* do sistema (KERNEL.ORG, 2011). Esse sub-sistema de contadores de performance pode ser acessados através da chamada de sistema *sys_perf_counter_open()* ou pelo conjunto de ferramentas disponibilizado pelo próprio sistema para acesso a essas informações (MELO, 2011).

O desenvolvimento das interfaces e ferramentas tem sido realizado por diversos desenvolvedores ligados ao *core* de desenvolvedores do Linux e essa integração é uma das grandes vantagens desse conjunto de ferramentas. Em algumas distribuições Linux como o Ubuntu, o pacote de ferramentas é chamado de **linux-tools**. No código fonte do *kernel* do Linux, essas ferramentas estão disponíveis no diretório *tools/perf*.

Na sequência são apresentadas algumas das ferramentas disponíveis no pacote **linux-tools**, com exemplos de utilização.

```
perf stat --repeat 1 date
Thu Feb  3 22:15:06 BRST 2011

 Performance counter stats for 'date':

       3.185250  task-clock-msecs         #      0.794 CPUs
              1  context-switches         #      0.000 M/sec
              0  CPU-migrations           #      0.000 M/sec
            212  page-faults              #      0.067 M/sec
        2513625  cycles                   #    789.145 M/sec
        1632767  instructions             #      0.650 IPC
          44277  cache-references         #     13.901 M/sec
           2622  cache-misses             #      0.823 M/sec

    0.004012171  seconds time elapsed
```

Figura 13: Exemplo de execução do *perf-stat*
Fonte: Autoria própria

A ferramenta *perf-stat* permite selecionar os contadores a serem monitorados durante a execução da aplicação, e gera os relatórios exibidos nas figuras 13 e 14. No primeiro caso, o *perf-stat* foi executado apenas uma vez e para o segundo caso, foram 5 execuções sucessivas do aplicativo, ativadas com a opção *repeat*. No caso de execuções sucessivas, a ferramenta disponibiliza no relatório o desvio padrão calculado sobre as **n** amostras obtidas com as execuções.

Antes de iniciar a análise da aplicação, é necessário identificar quais os contadores e eventos serão monitorados. Como esses recursos são dependentes da arquitetura de *hardware* em uso, pode-se consultar o manual do fabricante do processador ou então utilizar-se de ferramentas como a *perf-list* para a listagem das opções possíveis.

```
perf stat --repeat 5 date
Thu Feb  3 22:14:58 BRST 2011
Thu Feb  3 22:14:58 BRST 2011
Thu Feb  3 22:14:58 BRST 2011
Thu Feb  3 22:14:58 BRST 2011
Thu Feb  3 22:14:58 BRST 2011

 Performance counter stats for 'date' (5 runs):

       1.815635  task-clock-msecs         #      0.865 CPUs    ( +-  19.706%)
              0  context-switches         #      0.000 M/sec   ( +-  61.237%)
              0  CPU-migrations           #      0.000 M/sec   ( +-    -nan%)
            212  page-faults              #      0.117 M/sec   ( +-   0.000%)
        2635797  cycles                   #   1451.721 M/sec   ( +-   1.203%)
        1632482  instructions             #      0.619 IPC     ( +-   0.245%)
          42579  cache-references         #     23.451 M/sec   ( +-   1.026%)
           2511  cache-misses             #      1.383 M/sec   ( +-   4.442%)

    0.002098633  seconds time elapsed   ( +-  19.494% )
```

Figura 14: Exemplo de execução múltipla do *perf-stat*
Fonte: Autoria própria

A figura 15 apresenta um exemplo de listagem gerado com o perf-list a partir de um *notebook* equipado com um processador Intel Core 2 Duo.

```
List of pre-defined events (to be used in -e):

  cpu-cycles OR cycles                       [Hardware event]
  instructions                               [Hardware event]
  cache-references                           [Hardware event]
  cache-misses                               [Hardware event]
  branch-instructions OR branches            [Hardware event]
  branch-misses                              [Hardware event]
  bus-cycles                                 [Hardware event]

  cpu-clock                                  [Software event]
  task-clock                                 [Software event]
  page-faults OR faults                      [Software event]
  minor-faults                               [Software event]
  major-faults                               [Software event]
  context-switches OR cs                     [Software event]
  cpu-migrations OR migrations               [Software event]

  L1-dcache-loads                            [Hardware cache event]
  L1-dcache-load-misses                      [Hardware cache event]
  L1-dcache-stores                           [Hardware cache event]

  rNNN                                       [raw hardware event descriptor]

  i915:i915_gem_object_create                [Tracepoint event]
  i915:i915_gem_object_bind                  [Tracepoint event]
```

Figura 15: Alguns eventos disponíveis no *perf*
Fonte: Autoria própria

Um relatório gerado com a ferramenta *ophelp* é apresentado na figura 16. O *ophelp* é disponibilizado com a ferramenta de análise Oprofile, que possui funcionalidades semelhantes às fornecidas pelo **linux-tools**, porém requer o carregamento de um módulo de *kernel* para fazer o interfaceamento entre o *hardware* e a ferramenta utilizada pelo usuário (MELO, 2011). Neste caso, o *ophelp* foi utilizado por apresentar um listagem mais detalhada sobre os eventos e suas características.

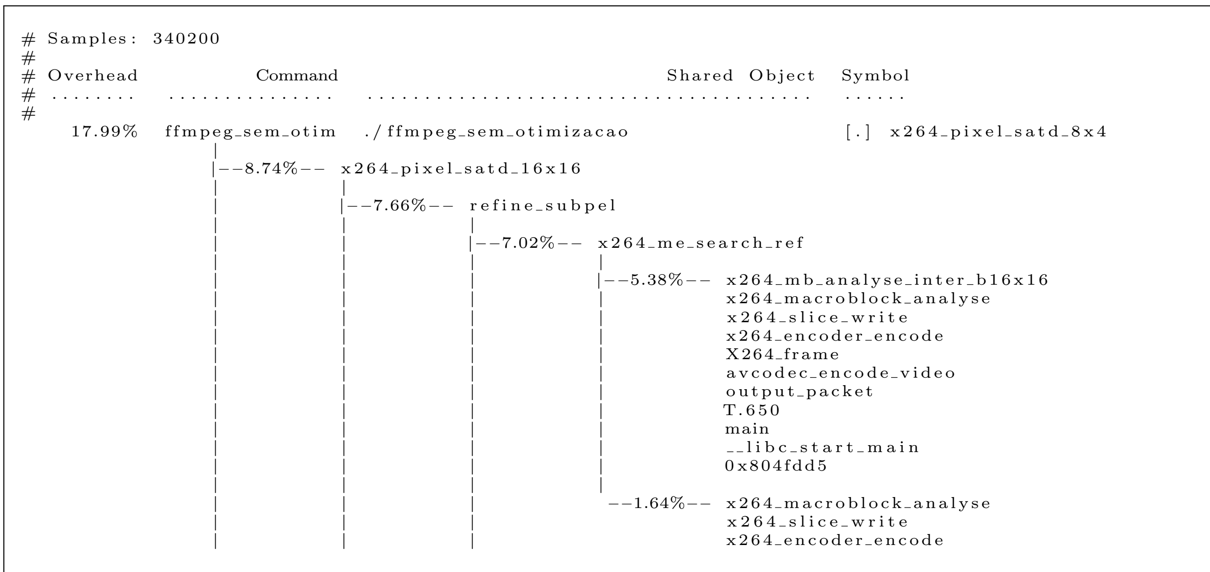
Caso o evento de *hardware* desejado não esteja listado no relatório do *perf-list*, é possível fazer a seleção do evento no formato *raw* através da sintaxe **perf stat -e rxxx**, ao invés da utilização de um nome de evento predefinido como **perf stat -e cpu-**

```
ophelp
oprofile: available events for CPU type "Core 2"

See Intel Architecture Developer's Manual Volume 3B, Appendix A and
Intel Architecture Optimization Reference Manual (730795-001)

CPU_CLK_UNHALTED: (counter: all)
        Clock cycles when not halted (min count: 6000)
        Unit masks (default 0x0)
        ----------
        0x00: Unhalted core cycles
        0x01: Unhalted bus cycles
        0x02: Unhalted bus cycles of this core while the other core is
        halted
INST_RETIRED_ANY_P: (counter: all)
        number of instructions retired (min count: 6000)
MISALIGN_MEM_REF: (counter: all)
        number of misaligned data memory references (min count: 500)
CYCLES_DIV_BUSY: (counter: all)
        cycles divider is busy (min count: 1000)
BUS_IO_WAIT: (counter: all)
        IO requests waiting in the bus queue (min count: 500)
        Unit masks (default 0x40)
        ----------
        0xc0: All cores
        0x40: This core
BR_INST_DECODED: (counter: all)
        number of branch instructions decoded (min count: 500)
```

Figura 16: Alguns eventos disponíveis no *Oprofile*
Fonte: Autoria própria

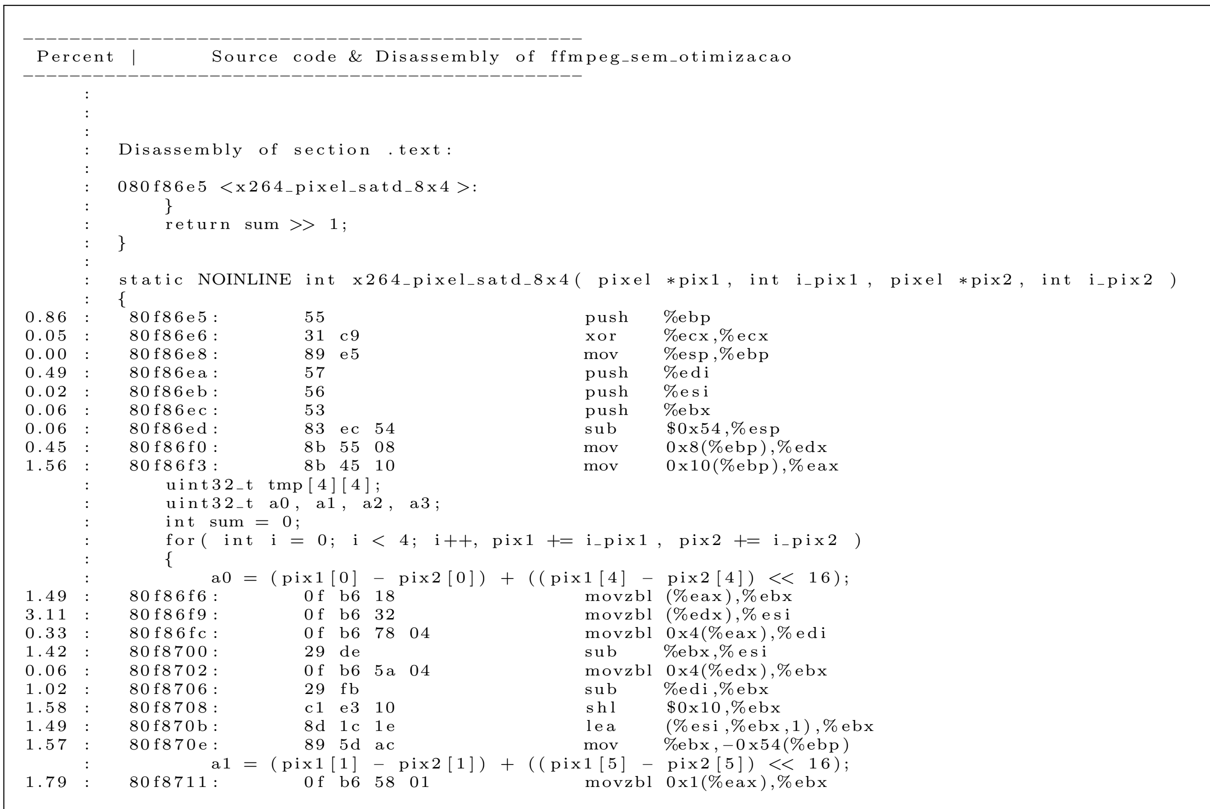
cycles. Para selecionar o evento, é necessário consultar a documentação de referência do processador em uso ou o relatório do *ophelp* para codificar o evento no formato <UMASK VALUE><EVENT NUM>.

As funções que causam maior impacto na execução da aplicação podem ser identificadas através do relatório de *call graph* e da saída do comando perf-annotate conforme os exemplos apresentados nas figuras 17 e 18. A partir do *call graph*, é possível identificar os pontos que demandam maior tempo de execução em toda a aplicação e a partir da análise do resultado do perf-annotate, é possível selecionar a melhor abordagem a ser utilizada na melhoria da função selecionada.



```
# Samples: 340200
#
# Overhead          Command                                   Shared Object   Symbol
# ........   ...............   ....................................   ......
#
   17.99%   ffmpeg_sem_otim   ./ffmpeg_sem_otimizacao                  [.] x264_pixel_satd_8x4
                  |
                  |--8.74%-- x264_pixel_satd_16x16
                  |          |
                  |          |--7.66%-- refine_subpel
                  |          |          |
                  |          |          |--7.02%-- x264_me_search_ref
                  |          |          |          |
                  |          |          |          |--5.38%-- x264_mb_analyse_inter_b16x16
                  |          |          |          |          x264_macroblock_analyse
                  |          |          |          |          x264_slice_write
                  |          |          |          |          x264_encoder_encode
                  |          |          |          |          X264_frame
                  |          |          |          |          avcodec_encode_video
                  |          |          |          |          output_packet
                  |          |          |          |          T.650
                  |          |          |          |          main
                  |          |          |          |          __libc_start_main
                  |          |          |          |          0x804fdd5
                  |          |          |          |
                  |          |          |           --1.64%-- x264_macroblock_analyse
                  |          |          |                     x264_slice_write
                  |          |          |                     x264_encoder_encode
```

Figura 17: Exemplo de relatório gerado com o comando *perf-report –call-graph*
Fonte: Autoria própria



```
------------------------------------------------
 Percent |      Source code & Disassembly of ffmpeg_sem_otimizacao
------------------------------------------------
         :
         :
         :
         :   Disassembly of section .text:
         :
         :   080f86e5 <x264_pixel_satd_8x4>:
         :       }
         :       return sum >> 1;
         :   }
         :
         :   static NOINLINE int x264_pixel_satd_8x4( pixel *pix1, int i_pix1, pixel *pix2, int i_pix2 )
         :   {
    0.86 :    80f86e5:       55                       push   %ebp
    0.05 :    80f86e6:       31 c9                    xor    %ecx,%ecx
    0.00 :    80f86e8:       89 e5                    mov    %esp,%ebp
    0.49 :    80f86ea:       57                       push   %edi
    0.02 :    80f86eb:       56                       push   %esi
    0.06 :    80f86ec:       53                       push   %ebx
    0.06 :    80f86ed:       83 ec 54                 sub    $0x54,%esp
    0.45 :    80f86f0:       8b 55 08                 mov    0x8(%ebp),%edx
    1.56 :    80f86f3:       8b 45 10                 mov    0x10(%ebp),%eax
         :       uint32_t tmp[4][4];
         :       uint32_t a0, a1, a2, a3;
         :       int sum = 0;
         :       for( int i = 0; i < 4; i++, pix1 += i_pix1, pix2 += i_pix2 )
         :       {
         :           a0 = (pix1[0] - pix2[0]) + ((pix1[4] - pix2[4]) << 16);
    1.49 :    80f86f6:       0f b6 18                 movzbl (%eax),%ebx
    3.11 :    80f86f9:       0f b6 32                 movzbl (%edx),%esi
    0.33 :    80f86fc:       0f b6 78 04              movzbl 0x4(%eax),%edi
    1.42 :    80f8700:       29 de                    sub    %ebx,%esi
    0.06 :    80f8702:       0f b6 5a 04              movzbl 0x4(%edx),%ebx
    1.02 :    80f8706:       29 fb                    sub    %edi,%ebx
    1.58 :    80f8708:       c1 e3 10                 shl    $0x10,%ebx
    1.49 :    80f870b:       8d 1c 1e                 lea    (%esi,%ebx,1),%ebx
    1.57 :    80f870e:       89 5d ac                 mov    %ebx,-0x54(%ebp)
         :           a1 = (pix1[1] - pix2[1]) + ((pix1[5] - pix2[5]) << 16);
    1.79 :    80f8711:       0f b6 58 01              movzbl 0x1(%eax),%ebx
```

Figura 18: Exemplo de relatório gerado com o comando *perf-annotate*
Fonte: Autoria própria

2.6 QUALIDADE DO VÍDEO DIGITAL

Em pesquisas com codificadores de vídeo, as métricas para avaliação da qualidade são extremamente importantes para a validação dos resultados gerados. As métricas mais utilizadas podem ser classificadas em subjetivas e objetivas.

A avaliação subjetiva é feita por observadores que, após assistirem a uma sequência de imagens ou vídeos, respondem a questionários sobre o que foi apresentado. As respostas de todos os observadores são então compiladas para a obtenção da avaliação geral do vídeo ou imagens apresentados. Esse tipo de métrica leva em consideração o impacto causado sobre o observador, a influência do conteúdo do vídeo, do ambiente em que é apresentado, os conhecimentos técnicos do observador e diversos outros parâmetros subjetivos. Maiores informações sobre avaliação subjetiva de vídeo podem ser obtidas em (SILVA, 2009) e nas recomendações ITU sobre avaliação subjetiva de qualidade (ITU, 2009, 2008).

A avaliação objetiva permite determinar a qualidade do vídeo através de parâmetros obtidos a partir de modelos matemáticos. Dentre as métricas objetivas mais utilizadas estão a PSNR, a MSE e a SSIM (SHEIKH; BOVIK, 2006). Em (ALBINI, 2009), são apresentados alguns dos problemas mais comuns encontrados nos vídeos digitais, bem como algumas das ferramentas utilizadas em análise objetiva de vídeo.

O erro quadrático médio ou MSE é calculado pela equação (7). Para um quadro (de vídeo ou imagem) com m linhas e n colunas, $f(i,j)$ corresponde ao valor do *pixel* na imagem original e $F(i,j)$ ao valor do *pixel* no quadro sob avaliação.

$$MSE = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n}[(f(i,j) - F(i,j))]^2}{m \; x \; n} \tag{7}$$

Para a avaliação da qualidade, quanto menor o valor de MSE, mais o quadro em análise se assemelha ao quadro original. Assim, para dois quadros iguais, o valor do MSE deve ser zero.

A métrica PSNR é equivalente à métrica MSE e é considerada mais conveniente por usar a escala logarítmica, como pode ser visto na equação (8). Nessa equação, $MaxVal = 2^n - 1$ e corresponde ao valor máximo da componente de vídeo, sendo n a quantidade de *bits* usada na representação da componente. No caso de componentes representadas em 8 bits, MaxVal tem o valor 255.

Comparando-se o valor do PSNR de duas imagens ou sequências de vídeo, o maior valor indica uma melhor qualidade objetiva.

$$PSNR = 10.log_{10}\frac{MaxVal^2}{MSE} \quad (8)$$

A métrica SSIM foi desenvolvida com o objetivo de aproximar os resultados da métrica ao da qualidade visual percebida pelo sistema visual humano, o que não pode ser atingido com as tradicionais métricas PSNR e MSE. Essa métrica procura determinar a similaridade dos quadros a partir da similaridade da luminância, do contraste e da similaridade estrutural. A equação para o cálculo da métrica é mais complexa que a das métricas anteriores e maiores informações sobre a métrica e os diversos conceitos envolvidos em sua definição podem ser obtidos em (WANG et al., 2004).

As três métricas citadas são classificadas como métricas de referência total (*full-reference*) pois utilizam todo o vídeo ou imagem durante a avaliação. Informações adicionais sobre essas e outras métricas objetivas para qualidade de vídeo podem ser obtidas em (MSU, 2011, Metrics Info)

As métricas subjetivas tendem a gerar resultados mais próximos da sensação percebida pelos observadores porém necessitam de um tempo maior para a obtenção dos resultados finais da avaliação e exigem um grande número de avaliadores para melhorar a precisão do resultado. Os resultados das métricas objetivas permitem uma comparação mais direta entre sequências de vídeo diferentes e tem como principal vantagem a possibilidade de serem utilizadas na avaliação de vídeo em tempo real. Um exemplo de aplicação que utiliza diversas métricas objetivas para a avaliação da qualidade de vídeos e imagens é a ferramenta MSU, disponível em (MSU, 2011).

2.7 CONCLUSÃO

Alguns conceitos importantes para a melhoria da performance de aplicações foram abordados neste capitulo, incluindo recursos da arquitetura x86, tópicos relacionados à intercomunicação entre a CPU e os dispositivos periféricos e ferramentas para análise da performance. De um modo geral, a comunicação entre a CPU e os periféricos é mais lenta e apresenta diversas limitações entre as quais pode-se citar a interface de comunicação utilizada (PCI, PCIe, USB etc) e a banda disponível entre os *chips* que interconectam os dispositivos, como a banda entre a CPU e a *NorthBridge* ou a conexão com a *SouthBridge*. Essas interconexões podem mudar entre as diferentes versões e famílias de processadores

como foi o caso da família Intel Core i7 que integrou a *NorthBridge* ao encapsulamento do processador, aumentando assim a taxa de transferência entre a CPU e dispositivos críticos como as memórias DDR e a interface PCIe. No caso de processadores produzidos pela AMD, as interconexões entre a CPU e os *chipsets* são diferentes mas as mesmas limitações continuam presentes.

Outro ponto abordado foi a diferença na largura de barramento entre as diversas famílias de processadores, principalmente os utilizados em dispositivos embarcados que, por questões principalmente de tamanho, preço e consumo de energia, tendem a fazer uso de barramento mais estreitos e consequentemente atingem taxas de transferências mais baixas.

Na seção referente à arquitetura das GPUs, foram apresentados os aspectos gerais desse tipo de dispositivo, que tem sido empregado em pesquisas científicas devido a sua alta capacidade de processamento. Dentre as principais características, destacam-se a grande quantidade de núcleos de processamento, que permitem um alto paralelismo na execução das aplicações e a alta taxa de transferência entre o núcleo da GPU e sua memória interna, devido a maior largura de banda do barramento utilizado. Como pontos principais que devem ser levados em consideração na utilização desses dispositivos, está a largura de banda disponível entre a CPU e a GPU, disponibilizada pela interface PCIe e a necessidade de paralelizar as operações da aplicação de modo a atingir o desempenho máximo da arquitetura.

Nos tópicos relacionados à otimização de *software* foram abordados temas referentes às técnicas mais dependentes da arquitetura de *hardware* utilizada, como o uso de linguagem *assembly*, disponível para todos os processadores, e instruções de máquina SIMD, disponíveis apenas em algumas linhas de processadores. A principal desvantagem dessas técnicas é a dificuldade de utilização das otimizações implementadas em arquiteturas de *hardware* diferentes e em alguns casos, até mesmo em processadores de gerações diferentes de uma mesma família.

As otimizações específicas para os codificadores, mais focadas em codificadores H.264 e seus predecessores, envolvem tanto a otimização da arquitetura da aplicação, de modo a aproveitar melhor os recursos de *hardware*, quanto a otimização do acesso à memória e a otimização dos algoritmos utilizados no processamento e análise do vídeo. A seleção de algoritmos diferenciados para determinados módulos dos codificadores e decodificadores já está disponível em muitos dos codificadores atualmente.

As ferramentas de *software profiling* são muito importantes durante o processo de

análise pois permitem a identificação de pontos críticos para o desempenho do *software*. Essas ferramentas, aliadas a recursos disponíveis nos processadores atuais, como os HPC são ferramentas indispensáveis.

A avaliação da qualidade do vídeo codificado é de extrema importância nos casos em que se pretende estudar os codificadores e decodificadores de vídeo. Várias ferramentas como a MSU (MSU, 2011) permitem verificar se em relação a um codificador ou decodificador de referência, ocorreram degradações no vídeo produzido, o que em algumas situações poderiam invalidar os ganhos obtidos.

3 DESENVOLVIMENTO

O desenvolvimento e resultados obtidos com o estudo e aplicação de técnicas para melhorar a performance de codificadores de vídeo são apresentados neste capítulo. Um conjunto de técnicas de otimização é apresentado no capítulo 2 e podem envolver alterações de algoritmo do codificador, utilização de instruções dependentes da arquitetura de *hardware*, caso das instruções MMX e SSE, e outras técnicas de implementação para *software* em geral. Para este trabalho, optou-se pela melhoria no acesso à memória como forma de aumentar a performance do codificador x264 (X264, 2011a). Um trabalho auxiliar foi desenvolvido com o objetivo de avaliar o impacto causado por diferentes modos de acesso à memória sobre o tempo de execução de uma aplicação simples com acesso intenso à memória externa e teve como objetivo principal a análise de técnicas de otimização e o desenvolvimento de um método de análise e validação que pudesse ser aplicado durante a otimização do codificador.

De acordo com (HENNESSY; PATTERSON, 2006), o tempo de execução da aplicação pode ser usado como medida da performance. Neste caso, a redução no tempo de execução de uma versão em relação a outra, indica uma melhora na performance. Para o caso do codificador de vídeo otimizado neste trabalho, além de considerar a melhora na performance da execução, os parâmetros objetivos de qualidade do vídeo codificado são levados em consideração para que o resultado das melhorias seja considerado válido.

Este capítulo está organizado da seguinte forma: Os equipamentos utilizados são indicados na seção 3.1. Na seção 3.2 é avaliado o impacto causado pelo modo de acesso à memória sobre o tempo de execução de uma aplicação simples e, finalmente, a seção 3.3 apresenta o desenvolvimento e resultados obtidos com a redução no acesso à memória para o codificador x264.

3.1 EQUIPAMENTOS UTILIZADOS

Os equipamentos utilizados nos estudos apresentados na sequência, são especificados abaixo :

1. *notebook1* : *Notebook* com processador Intel Core2 Duo CPU T7250 @ 2.00GHz e 2 Gb de memória RAM, com suporte a MMX, SSE e SSE2, executando o Sistema Operacional Linux Ubuntu 10.01;

2. *servidor1* : Servidor equipado com uma CPU Intel Core2 Quad 2.8GHz, 8 GB de memória RAM e uma placa de vídeo NVIDIA GeForce GTX 285 (1 GB), executando o Sistema Operacional Linux 64 bits ArchLinux. A placa de vídeo deste sistema, cujas especificações estão indicadas na tabela 14, foi utilizada como GPU durante os testes.

Tabela 14: Especificação da GPU utilizada

Processador		Memória	
Qtd. *cores* CUDA	240	*Clock* da interface	1242 MHz
		Configuração padrão	1 GB GDDR3
Clock do processador	1476 MHz	Largura da interface	512 bits
		Largura de banda	159,0 GB/s

Fonte: Adaptado de (NVI, 2011)

3.2 PERFORMANCE NO ACESSO À MEMÓRIA

Esta seção apresenta o estudo sobre o impacto do modo de acesso à memória sobre o tempo de execução de um aplicativo, permitindo também a comparação do resultado da execução em uma CPU tradicional em relação a uma GPU.

Este estudo foi desenvolvido para auxiliar na escolha da abordagem a ser utilizada na otimização do codificador x264, que é apresentada na seção 3.3. Os resultados obtidos podem ser utilizados como parâmetro para se estimar os valores máximos de redução no tempo de execução da aplicação, quando adotada a mesma técnica de minimização do acesso à memória.

3.2.1 IMPLEMENTAÇÃO

Um aplicativo em linguagem C foi desenvolvido com o objetivo de avaliar o tempo de execução de um conjunto de operações matemáticas simples usando diferentes padrões de acesso à memória e também executando as operações em uma GPU. O teste consistia na execução de 209.715.200 somas sobre operandos de 8 bits, sendo os valores utilizados limitados ao máximo de 127, evitando assim *overflow*. Para o acesso diferenciado de cada modo, foram criados os tipos de dados apresentados na figura 19 com os quais era possível usar ponteiros para diferentes tipos de dados no acesso ao *buffer* de resultados e as operações matemáticas eram executadas com a utilização do ponteiro para o tipo uint8_t.

As execuções foram repetidas cinco vezes para cada modo selecionado. Na CPU foram utilizados os modos em 8, 16, 32 e 64 bits e no caso do servidor, além da execução com diferentes modos de acesso à memória, o aplicativo também foi executado na GPU. Finalizadas as operações de soma, o aplicativo executava uma verificação do resultado da soma através da comparação do resultado obtido com o resultado esperado, usando o mesmo tipo de acesso usado para as operações de soma, isto é, se a operação foi executada com acessos em 8 bits, a validação do resultado era executada com acessos de 8 bits. Para o caso da execução das somas na GPU, manteve-se a forma das operações porém a função responsável pela execução das operações matemáticas precisou ser reescrita, de acordo com a API CUDA. A verificação do resultado da soma na GPU foi executada na CPU utilizando-se acessos de 8 bits.

```
typedef union u16_u8_union {
        uint8_t *u8;
        uint16_t *u16;
} u16_u8_t;

typedef union u32_u8_union {
        uint8_t *u8;
        uint32_t *u32;
} u32_u8_t;

typedef union u64_u8_union {
        uint8_t *u8;
        uint64_t *u64;
} u64_u8_t;
```

Figura 19: Tipos de dados para os modos de acesso à memória
Fonte: Autoria própria

Em linhas gerais, os procedimentos executados pelo aplicativo para cada teste eram os seguintes:

* Alocação de um *buffer* de 200 Mbytes;

* Execução de 200 Mega operações de soma e armazenamento do resultado no *buffer* alocado inicialmente;

* Verificação do resultado das operações através da comparação com uma matriz de resultados previamente calculados.

O aplicativo era executado pela ferramenta **perf-stat** disponível em versões recentes do Sistema Operacional Linux. Ao final, a ferramenta disponibilizava um relatório com os valores dos contadores de performance selecionados, o tempo de execução e o desvio padrão calculado para cada item do relatório.

Os testes foram executados nos equipamentos *notebook1* e servidor1 listados na seção 3.1 e o código-fonte da aplicação é apresentado nas figuras 20, 21 e 22.

Para a CPU, a aplicação foi executada em um único processo, apesar de as CPUs utilizadas serem multi-processadas. Com essa abordagem eliminou-se a necessidade de implementar mecanismos de sincronização e gerenciamento das *threads*, mantendo-se o foco na análise do acesso ao barramento externo, que só pode ser acessado por um processo de cada vez.

A abordagem adotada para a GPU foi diferente e optou-se pela utilização de uma *thread* para cada uma das operações de soma visto que o paralelismo e o uso de grandes quantidades de *threads* é uma das características das implementações para as GPU, pois a arquitetura provê e gerencia esse tipo de recurso. A GPU utilizada permite a execução de até 30.720 *threads* concorrentes e esse número é determinado pela equação (9), sendo que $num_SM = 30$ e $threads_por_sm = 1.024$ para a GPU em questão (NVIDIA, 2008).

$$max_threads = num_SM \; x \; threads_por_sm \tag{9}$$

O código-fonte da aplicação adaptada para a execução na GPU é apresentado na figura 23 e precisou ser compilado com as ferramentas especificas, fornecidas pelo fabricante da GPU, e a execução ocorreu conforme indicado no capítulo de fundamentação, sub-seção 2.1.4.

```
1 #include <stdio.h>
2 #include <stdint.h>
3 #include <stdlib.h>
4
5 typedef union u16_u8_union {
6     uint8_t *u8;
7     uint16_t *u16;
8 } u16_u8_t;
9
10 typedef union u32_u8_union {
11     uint8_t *u8;
12     uint32_t *u32;
13 } u32_u8_t;
14
15 typedef union u64_u8_union {
16     uint8_t *u8;
17     uint64_t *u64;
18 } u64_u8_t;
19
20 unsigned char oper_a[16]              = { 2, 4, 6, 8,10,12,14,16,18,20,22,24,26,28,30,32};
21 unsigned char oper_b[16]              = { 1, 3, 5, 7, 9,11,13,15,17,19,21,23,25,27,29,31};
22 unsigned char res_soma_a_b[16]  = { 3, 7,11,15,19,23,27,31,35,39,43,47,51,55,59,63};
23
24 void soma8( void *data, uint32_t size )
25 {
26     uint8_t *data8 = (uint8_t*) data;
27     uint8_t *data8End = data8 + size;
28     int cnt=0;
29
30     while ( data8 != data8End ) {
31         *data8++ = oper_a[cnt] + oper_b[cnt];
32         if ( cnt > 14 ) { // se >- 15, zera
33             cnt=0;
34         } else {
35             cnt++;
36         }
37     }
38 }
39
40 void check_soma8( void *data, uint32_t size )
41 {
42     uint8_t *data8 = (uint8_t*) data;
43     uint8_t *data8End = data8 + size;
44     unsigned int cnt=0, ok_cnt=0, nok_cnt=0;
45
46     while ( data8 != data8End ) {
47
48         if ( *data8++ == res_soma_a_b[cnt] ) {
49             ok_cnt++;
50         } else {
51             nok_cnt++;
52         }
53
54         if ( cnt > 14 ) { // se >= 15, zera
55             cnt=0;
56         } else {
57             cnt++;
58         }
59     }
60     printf("Check 8: OK %d NOK %d\n",ok_cnt, nok_cnt);
61 }
62
63 void soma16( void *data, uint32_t size )
64 {
65     u16_u8_t data_ptr,data_ptr_end;
66     unsigned int cnt=0;
67
68     data_ptr.u16 = (uint16_t *)data;
69     data_ptr_end.u16 = (uint16_t *)data + (size >> 1); /* soma o size dividido por 2 */
70
71     while ( data_ptr.u16 != data_ptr_end.u16 ) {
72
73         data_ptr.u8[0] = oper_a[cnt] + oper_b[cnt];
74         data_ptr.u8[1] = oper_a[cnt+1] + oper_b[cnt+1];
75
76         if ( cnt > 13 ) // se >= 14, zera
77             cnt=0;
78         } else {
79             cnt+=2;
80         }
81         *data_ptr.u16++;
82     }
83 }
```

Figura 20: Código-fonte da aplicação utilizada - 1/3
Fonte: Autoria própria

```
85 void check_soma16( void *data, uint32_t size )
86 {
87     u16_u8_t data_ptr, data_ptr_end;
88     unsigned int cnt=0, ok_cnt=0, nok_cnt=0;
89     data_ptr.u16 = (uint16_t *)data;
90     data_ptr_end.u16 = (uint16_t *)data + (size >> 1); /* soma o size dividido por 2 */
91     while ( data_ptr.u16 != data_ptr_end.u16 ) {
92         if ( data_ptr.u8[0] != res_soma_a_b[cnt] || data_ptr.u8[1] != res_soma_a_b[cnt+1] ) {
93             nok_cnt++;
94         else
95             ok_cnt++;
96
97         if ( cnt > 13 ) // se >= 14, zera
98             cnt=0;
99         else
100            cnt+=2;
101        *data_ptr.u16++;
102    }
103    printf("Check 16: OK %d NOK %d\n",ok_cnt, nok_cnt);
104 }
105
106 void soma32( void *data, uint32_t size )
107 {
108     u32_u8_t data_ptr, data_ptr_end;
109     unsigned int cnt=0;
110     data_ptr.u32 = (uint32_t *)data;
111     data_ptr_end.u32 = (uint32_t *)data + (size >> 2); /* soma o size dividido por 4 */
112     while ( data_ptr.u32 != data_ptr_end.u32 ) {
113         data_ptr.u8[0] = oper_a[cnt] + oper_b[cnt];
114         data_ptr.u8[1] = oper_a[cnt+1] + oper_b[cnt+1];
115         data_ptr.u8[2] = oper_a[cnt+2] + oper_b[cnt+2];
116         data_ptr.u8[3] = oper_a[cnt+3] + oper_b[cnt+3];
117         if ( cnt > 11 ) // se >= 12, zera
118             cnt=0;
119         else
120             cnt+=4;
121         *data_ptr.u32++;
122     }
123 }
124
125 void check_soma32( void *data, uint32_t size )
126 {
127     u32_u8_t data_ptr, data_ptr_end;
128     unsigned int cnt=0, ok_cnt=0, nok_cnt=0;
129     data_ptr.u32 = (uint32_t *)data;
130     data_ptr_end.u32 = (uint32_t *)data + (size >> 2); /* soma o size dividido por 4 */
131     while ( data_ptr.u32 != data_ptr_end.u32 ) {
132         if ( data_ptr.u8[0] != res_soma_a_b[cnt] || data_ptr.u8[1] != res_soma_a_b[cnt+1]
133         || data_ptr.u8[2] != res_soma_a_b[cnt+2] || data_ptr.u8[3] != res_soma_a_b[cnt+3] ) {
134             nok_cnt++;
135         } else {
136             ok_cnt++;
137         }
138         if ( cnt > 11 ) { // se >= 12, zera
139             cnt=0;
140         } else {
141             cnt+=4;
142         }
143         *data_ptr.u32++;
144     }
145     printf("Check 32: OK %d NOK %d\n",ok_cnt, nok_cnt);
146 }
147
148 void soma64( void *data, uint32_t size )
149 {
150     u64_u8_t data_ptr, data_ptr_end;
151     unsigned int cnt=0;
152     data_ptr.u64 = (uint64_t *)data;
153     data_ptr_end.u64 = (uint64_t *)data + (size >> 3); /* soma o size dividido por 8 */
154     while ( data_ptr.u64 != data_ptr_end.u64 ) {
155         data_ptr.u8[0] = oper_a[cnt] + oper_b[cnt];
156         data_ptr.u8[1] = oper_a[cnt+1] + oper_b[cnt+1];
157         data_ptr.u8[2] = oper_a[cnt+2] + oper_b[cnt+2];
158         data_ptr.u8[3] = oper_a[cnt+3] + oper_b[cnt+3];
159         data_ptr.u8[4] = oper_a[cnt+4] + oper_b[cnt+4];
160         data_ptr.u8[5] = oper_a[cnt+5] + oper_b[cnt+5];
161         data_ptr.u8[6] = oper_a[cnt+6] + oper_b[cnt+6];
162         data_ptr.u8[7] = oper_a[cnt+7] + oper_b[cnt+7];
163         if ( cnt > 7 ) { // se >= 8, zera
164             cnt=0;
165         } else {
166             cnt+=8;
167         }
168         *data_ptr.u64++;
169     }
170 }
```

Figura 21: Código-fonte da aplicação utilizada - 2/3
Fonte: Autoria própria

```
172 void check_soma64( void *data, uint32_t size )
173 {
174     u64_u8_t data_ptr, data_ptr_end;
175     unsigned int cnt=0,ok_cnt=0, nok_cnt=0;
176
177     data_ptr.u64 = (uint64_t *)data;
178     data_ptr_end.u64 = (uint64_t *)data + (size >> 3); /* soma o size dividido por 8 */
179
180     while ( data_ptr.u64 != data_ptr_end.u64 ) {
181
182         if ( data_ptr.u8[0] != res_soma_a_b[cnt] || data_ptr.u8[1] != res_soma_a_b[cnt+1]
183         || data_ptr.u8[2] != res_soma_a_b[cnt+2] || data_ptr.u8[3] != res_soma_a_b[cnt+3]
184         || data_ptr.u8[4] != res_soma_a_b[cnt+4] || data_ptr.u8[5] != res_soma_a_b[cnt+5]
185         || data_ptr.u8[6] != res_soma_a_b[cnt+6] || data_ptr.u8[7] != res_soma_a_b[cnt+7]) {
186             nok_cnt++;
187         } else {
188             ok_cnt++;
189         }
190
191         if ( cnt > 7 ) { // se >= 8, zera
192             cnt=0;
193         } else {
194             cnt+=8;
195         }
196         *data_ptr.u64++;
197     }
198     printf("Check 64: OK %d NOK %d\n",ok_cnt, nok_cnt);
199 }
200
201 int main(int argc, char *argv[])
202 {
203 #define BUFF_SIZE 200*1024*1024
204     char *buff;
205     int cmd;
206
207     if ( argc != 2 ) {
208         printf("Parametro invalido!\n");
209         return 1;
210     }
211
212     buff = malloc( BUFF_SIZE );
213     if ( buff == NULL ) {
214         printf("Nao foi possivel alocar memoria\n");
215         return 1;
216     }
217     cmd = atoi( argv[1] );
218
219     switch ( cmd ) {
220     case 8:
221         soma8( (void *)buff, BUFF_SIZE );
222         check_soma8( (void *)buff, BUFF_SIZE );
223         break;
224     case 16:
225         soma16( (void *)buff, BUFF_SIZE );
226         check_soma16( (void *)buff, BUFF_SIZE );
227         break;
228     case 32:
229         soma32( (void *)buff, BUFF_SIZE );
230         check_soma32( (void *)buff, BUFF_SIZE );
231         break;
232     case 64:
233         soma64((void *)buff, BUFF_SIZE );
234         check_soma64( (void *)buff, BUFF_SIZE );
235         break;
236     default:
237         printf("Opcao invalida\n");
238
239     }
240     free(buff);
241
242     return 0;
243 }
```

Figura 22: Código-fonte da aplicação utilizada - 3/3
Fonte: Autoria própria

```
1 unsigned char *dev_oper_a, *dev_oper_b;
2
3 /* Numero de threads por bloco (maximo 512 threads por bloco) */
4 dim3 dimBlock(512, 1);
5 /* Numero de blocos por grid */
6 dim3 dimGrid(640, 640);
7
8 /* kernel: funcao que e executada na GPU */
9 __global__ void soma8kernel(void *data, uint32_t size, unsigned char *dev_oper_a,
10                unsigned char *dev_oper_b ){
11         /* Cada bloco e identificado por um par (x,y) */
12         int uniqueBlockId = blockIdx.y * gridDim.x + blockIdx.x;
13         /* Cada thread e identificada por um par (x,y) */
14         int uniqueThreadId = uniqueBlockId * blockDim.y * blockDim.x +
15                 threadIdx.y * blockDim.x + threadIdx.x;
16         uint8_t *data8 = (uint8_t*) data;
17
18         /* se ID da thread for menor que o tamanho do buffer */
19         if (uniqueThreadId <= size) {
20                 /* transforma ID da thread em um indice entre 0 e 16 */
21                 int aux = uniqueThreadId/16;
22                 int auxIndex = uniqueThreadId - (aux*16);
23
24                 /* armazena o resultado */
25                 data8[uniqueThreadId] = dev_oper_a[auxIndex] + dev_oper_b[auxIndex];
26         }
27 }
28
29 int main_gpu(void)
30 {
31         /* buffer no host (CPU) */
32         char *buff;
33         /* buffer da GPU */
34         char *dev_buff;
35
36         buff = (char*) malloc( BUFF_SIZE );
37
38         /* aloca memoria na GPU para o buffer */
39         cudaMalloc((void**)&dev_buff, BUFF_SIZE);
40
41         /* aloca memoria na GPU para os arrays de operandos */
42         cudaMalloc((void**)&dev_oper_a, sizeof(oper_a));
43         cudaMalloc((void**)&dev_oper_b, sizeof(oper_b));
44
45         //copia o conteudo dos arrays de operandos para a GPU
46         cudaMemcpy(dev_oper_a, oper_a, sizeof(oper_a), cudaMemcpyHostToDevice);
47         cudaMemcpy(dev_oper_b, oper_b, sizeof(oper_b), cudaMemcpyHostToDevice);
48
49         /* Executa a funcao soma8kernel() na GPU */
50         soma8kernel<<<dimGrid,dimBlock>>>(dev_buff, BUFF_SIZE, dev_oper_a, dev_oper_b);
51
52         /* copia o buffer de resultados da GPU para o Host */
53         cudaMemcpy((void *)buff, dev_buff, BUFF_SIZE, cudaMemcpyDeviceToHost);
54
55         /* verifica os resultados */
56         check_soma8( (void *)buff, BUFF_SIZE );
57
58         /* libera memoria na gpu */
59         cudaFree(dev_oper_a);
60         cudaFree(dev_oper_b);
61
62         /* libera memoria no host */
63         free(buff);
64
65     return 0;
66 }
```

Figura 23: Código-fonte da aplicação adaptada para a GPU
Fonte: Autoria própria

3.2.2 RESULTADOS

As tabelas 15 e 16 apresentam respectivamente os resultados obtidos para o *notebook1* e servidor1. Na coluna **evento**, *cycles* é o número total de ciclos utilizados pela CPU, *bus-cycles* corresponde ao número de ciclos gastos no acesso ao barramento e o *tempo* é o tempo de execução da aplicação, sendo todos valores médios das 5 execuções. A coluna s corresponde ao desvio padrão das amostras obtidas nas 5 execuções de cada modo e foi calculado pela ferramenta perf-stat. A coluna Redução, refere-se à diminuição no valor médio de cada evento em relação ao acesso em 8 bits. Para o evento *bus-cycles*, a coluna α contém a percentagem que esse evento representa do valor total de ciclos (*cycles*).

O valor médio de ciclos totais e os ciclos gastos no acesso ao barramento para as execuções no *notebook1* e servidor1 são apresentados nas figuras 24 e 25 respectivamente. O tempo de execução médio para ambos os casos é apresentado na figura 26.

Tabela 15: Performance no acesso à memória - *notebook1*

Tipo de acesso [bits]	Evento	Valor Médio	α [%]	s [%]	Redução	
					Valor	[%]
8	*cycles*	4872450309	-	0,712	0	0
	bus-cycles	4479206722	91,929	0,749	0	0
	tempo [s]	2,182	-	0,484	0	0
16	*cycles*	3748600670	-	0,49	1123849639	23,065
	bus-cycles	3445574626	91,916	0,58	1033632096	23,076
	tempo [s]	1,674	-	0,084	0,508	23,286
32	*cycles*	3215305202	-	0,346	1657145107	34,011
	bus-cycles	2953421638	91,855	0,417	1525785084	34,064
	tempo [s]	1,441	-	0,116	0,741	33,978
64	*cycles*	2832653196	-	0,359	2039797113	41,864
	bus-cycles	2601745320	91,848	0,464	1877461402	41,915
	tempo [s]	1,269	-	0,086	0,913	41,852

Fonte: Autoria própria

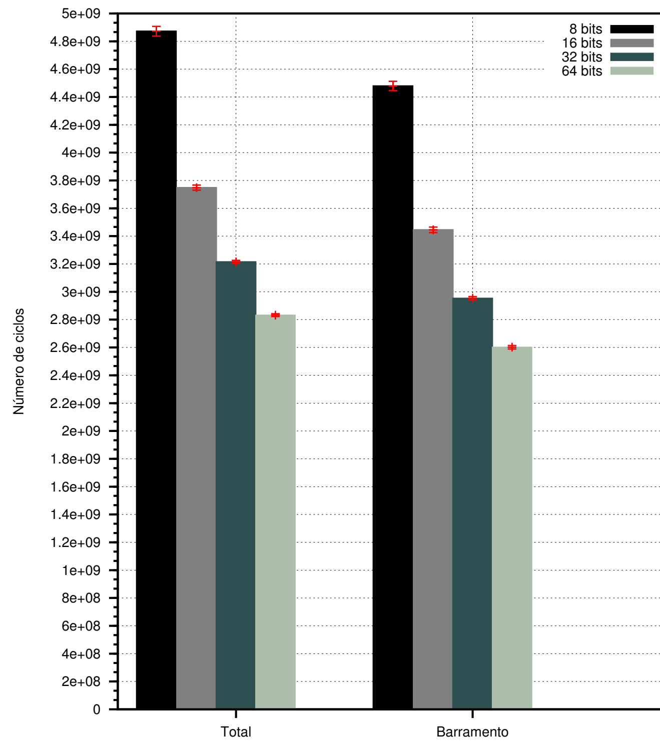


Figura 24: Ciclos contabilizados para as operações de soma no *notebook1*. Fonte: Autoria própria

Tabela 16: Performance no acesso à memória - servidor1

Tipo de acesso [bits]	Evento	Valor Médio	α [%]	s [%]	Redução Valor	Redução [%]
8	*cycles*	4614688257	-	0,137	0	0,000
	bus-cycles	576844275	12,5	0,137	0	0,000
	tempo [s]	1,728	-	0,138	0	0,000
16	*cycles*	3628533940	-	0,112	986154317	21,370
	bus-cycles	453575016	12,5	0,112	123269259	21,370
	tempo [s]	1,359	-	0,115	0,369	21,363
32	*cycles*	2966002631	-	0,154	1648685626	35,727
	bus-cycles	370757099	12,5	0,154	206087176	35,727
	tempo [s]	1,110	-	0,144	0,617	35,733
64	*cycles*	2611391081	-	0,091	2003297176	43,411
	bus-cycles	326431860	12,5	0,091	250412415	43,411
	tempo [s]	0,978	-	0,076	0,750	43,428
GPU	*cycles*	2547298724	-	0,142	2067389533	44,800
	bus-cycles	320111786	12,567	0,154	256732489	44,506
	tempo [s]	0,887	-	2,207	0,841	48,648

Fonte: Autoria própria

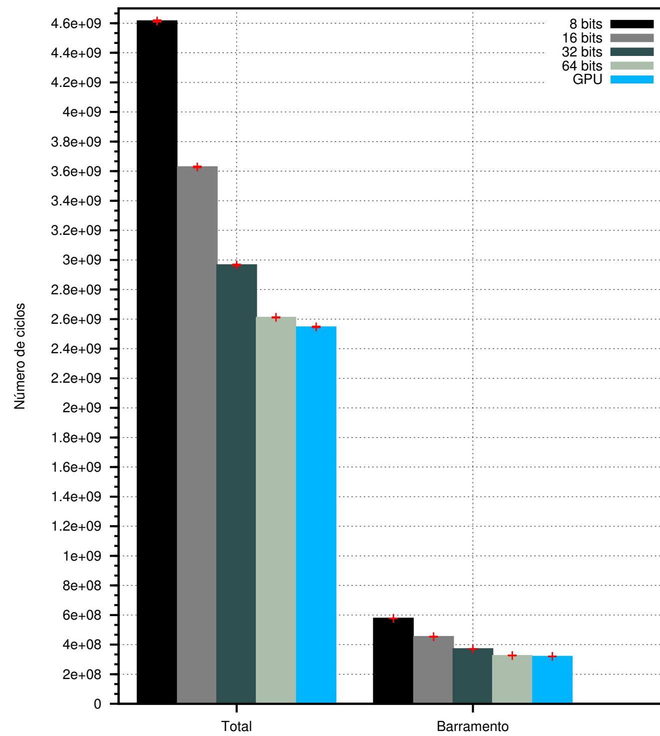


Figura 25: Ciclos contabilidados para as operações de soma no servidor1. Fonte: Autoria própria

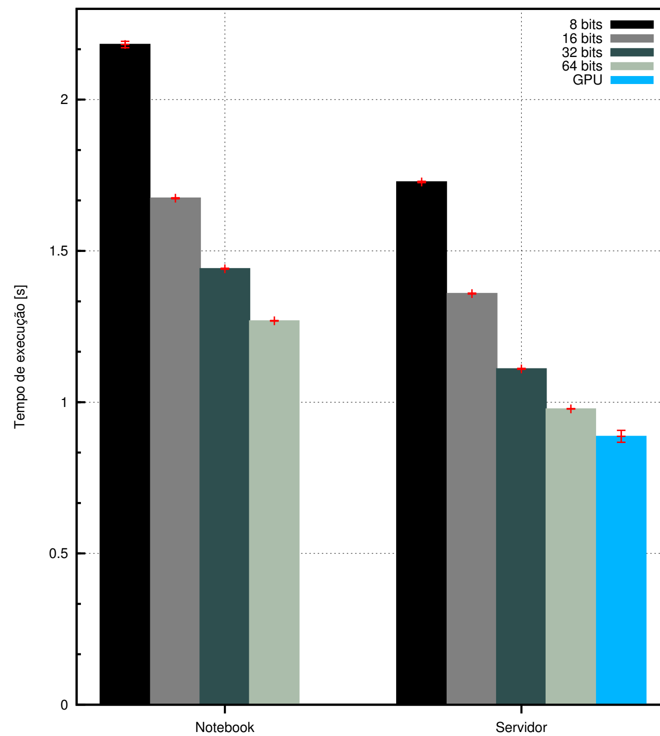


Figura 26: Tempo de execução para cada um dos equipamentos. Fonte: Autoria própria

3.3 REDUÇÃO DO ACESSO À MEMÓRIA PARA O CODIFICADOR X264

A redução no acesso à memória para o codificador x264 é apresentada nesta seção. Foram aplicadas técnicas de otimização para melhorar a performance do codificador, tendo como requisitos a manutenção da qualidade do vídeo codificado e o tamanho reduzido do codificador.

O codificador x264 é uma implementação em código aberto do padrão de codificação de vídeo H.264 e é disponibilizado sob a licença GNU GPL em (X264, 2011a). Esse codificador pode ser utilizado com o Sistema Operacional Linux em computadores de uso geral e em sistemas embarcados.

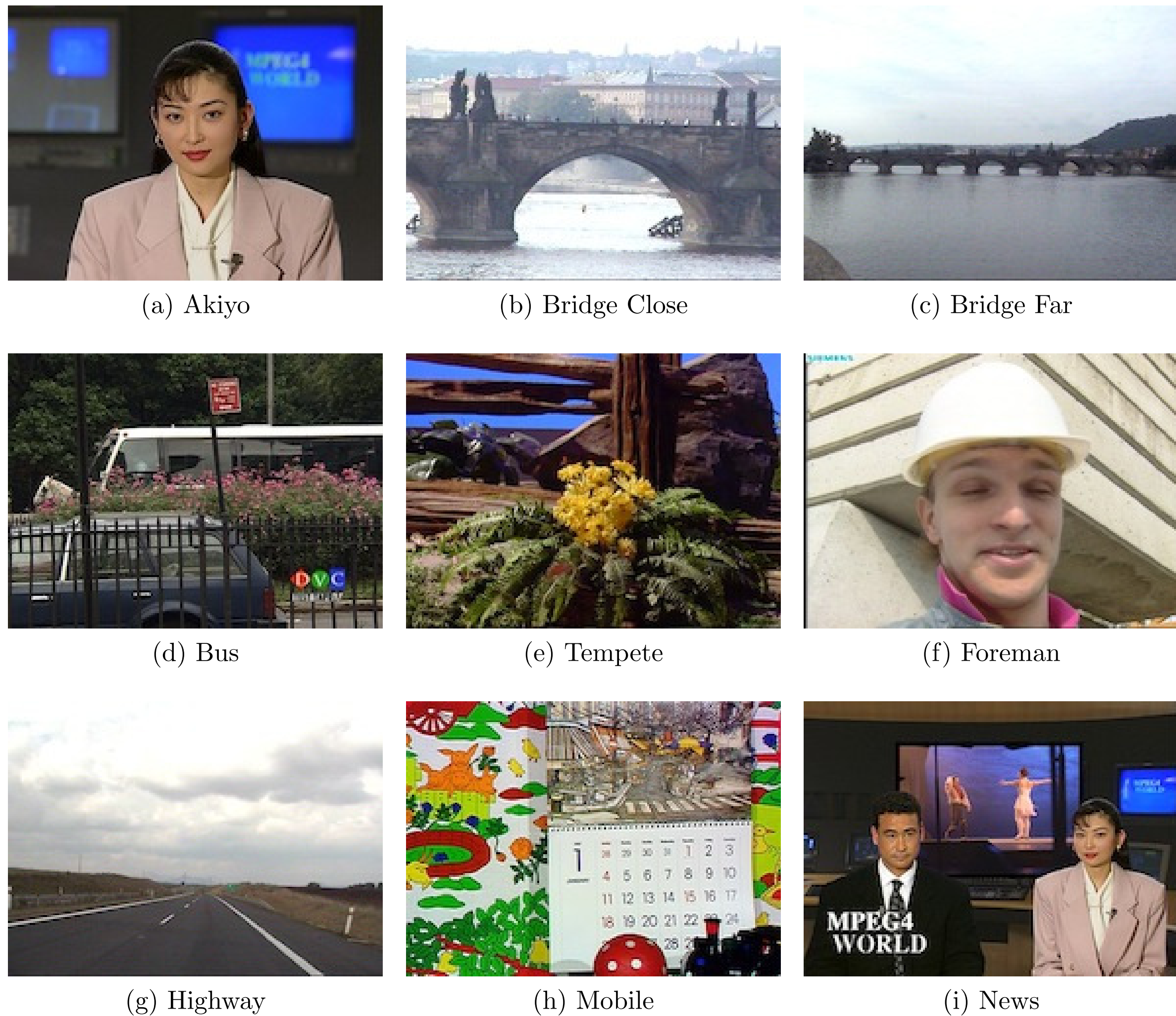
O *sofware* ffmpeg (FFMPEG, 2011a) foi utilizado como interface de usuário, obtendo o vídeo original a partir de arquivos e utilizando a biblioteca gerada durante a compilação do x264 para a codificação do vídeo no padrão H.264. O ffmpeg foi selecionado por permitir a obtenção da fonte de vídeo a partir de arquivos, placas de captura de vídeo e *streams* de rede além de disponibilizar um conjunto de ferramentas que podem ser aplicadas sobre o vídeo na entrada ou saída do processo de codificação.

3.3.1 IMPLEMENTAÇÃO

Os códigos-fonte do x264 e ffmpeg foram obtidos em (X264, 2011b) e (FFMPEG, 2011b) respectivamente. A figura 27 apresenta o quadro principal de cada um dos vídeos utilizados e as características são apresentadas na tabela 17. Os vídeos apresentam temas variados, permitindo avaliar a performance do codificador em diversas condições. Características como variação na quantidade de movimento em cada cena, maior ou menor quantidade de detalhes espaciais nas cenas podem interferir na qualidade e na velocidade da codificação. Todos os vídeos utilizados estavam no formato CIF e usando o espaço de cores YUV [Albini 2009].

Durante o desenvolvimento, as implementações das otimizações foram aplicadas sobre o código-fonte do x264. A biblioteca x264 gerada era então utilizada na compilação do ffmpeg e em seguida o processo de codificação era iniciado, sempre com a utilização da ferramenta perf para a análise dos pontos críticos ou obtenção dos dados sobre a performance do codificador.

Para gerar a biblioteca do x264, o *bit-depth* foi configurado para 8 (as amostras do vídeo são tratadas como sendo de 8 bits). O número de opções e parâmetros de con-



(a) Akiyo (b) Bridge Close (c) Bridge Far
(d) Bus (e) Tempete (f) Foreman
(g) Highway (h) Mobile (i) News

Figura 27: Imagens (quadros) obtidas dos vídeos utilizados
Fonte: (ASU, 2011)

Tabela 17: Características dos vídeos utilizados

Vídeo	Formato dos dados	Formato visual	Tamanho [*bytes*]	Quantidade quadros
Akiyo	4:2:0 YUV	CIF	45.619.200	300
Bridge Close	4:2:0 YUV	CIF	304.128.000	2001
Bridge Far	4:2:0 YUV	CIF	319.486.464	2101
Bus	4:2:0 YUV	CIF	22.809.600	150
Foreman	4:2:0 YUV	CIF	45.619.200	300
Highway	4:2:0 YUV	CIF	304.128.000	2000
Mobile	4:2:0 YUV	CIF	45.619.200	300
News	4:2:0 YUV	CIF	45.619.200	300
Tempete	4:2:0 YUV	CIF	39.536.640	260

Fonte: Autoria própria

figuração do ffmpeg foi reduzido ao máximo, sendo habilitados e compilados apenas os utilizados no ambiente de codificação. Essa medida foi tomada para reduzir ao máximo o tamanho do codificador gerado, de modo que ele possa ser utilizado em sistemas embarcados, onde os recursos de armazenamento são limitados. A opção de compilação para redução do tamanho do objeto gerado foi selecionada tanto para o x264 quanto para o ffmpeg.

O processo de análise e implementação seguiu o fluxograma indicado na figura 29.

```
typedef union pixel_64_u {
        uint8_t *u8;
        uint64_t *u64;
} pixel_64_t;
```

Figura 28: Tipo de dado para o acesso à memória
Fonte: Autoria própria

O *call-graph* foi obtido com o comando **perf report -g graph ...** e um exemplo de relatório é apresentado na figura 31.

A seleção da função a ser otimizada foi realizada a partir do conjunto de funções com maior percentagem de tempo de execução. Esse conjunto de funções foi obtido a partir do relatório *callgraph*. No exemplo citado, as funções **get_ref** e **x264_pixel_satd_8x4** seriam as candidatas a otimização.

Com o auxílio do relatório gerado pelo comando **perf annotate** a função selecionada era analisada, com o intuito de identificar a melhor abordagem a ser utilizada na otimização. Esse relatório apresenta uma indicação da percentagem atribuída a cada instrução em *assembly* da função em análise. No mesmo relatório são apresentados o código em linguagem C e o resultado equivalente em *assembly*. Um exemplo do relatório gerado pelo **perf annotate** pode ser visto na figura 30.

Após a etapa de implementação, o resultado foi validado com o uso do comando **perf stat**, que a partir de uma lista de eventos a serem monitorados, passados como parâmetro, produz um relatório equivalente ao da figura 32.

As métricas objetivas de qualidade do vídeo PSNR, MSE e SSIM foram obtidas com a utilização da ferramenta MSU (MSU, 2011) e foram utilizadas como condição para aceitar as otimizações implementadas.

As otimizações foram aplicadas sobre as funções utilizadas no cálculo da SATD e SAD, e sobre as funções pixel_avg e mc_chroma. Para controlar a ativação de cada

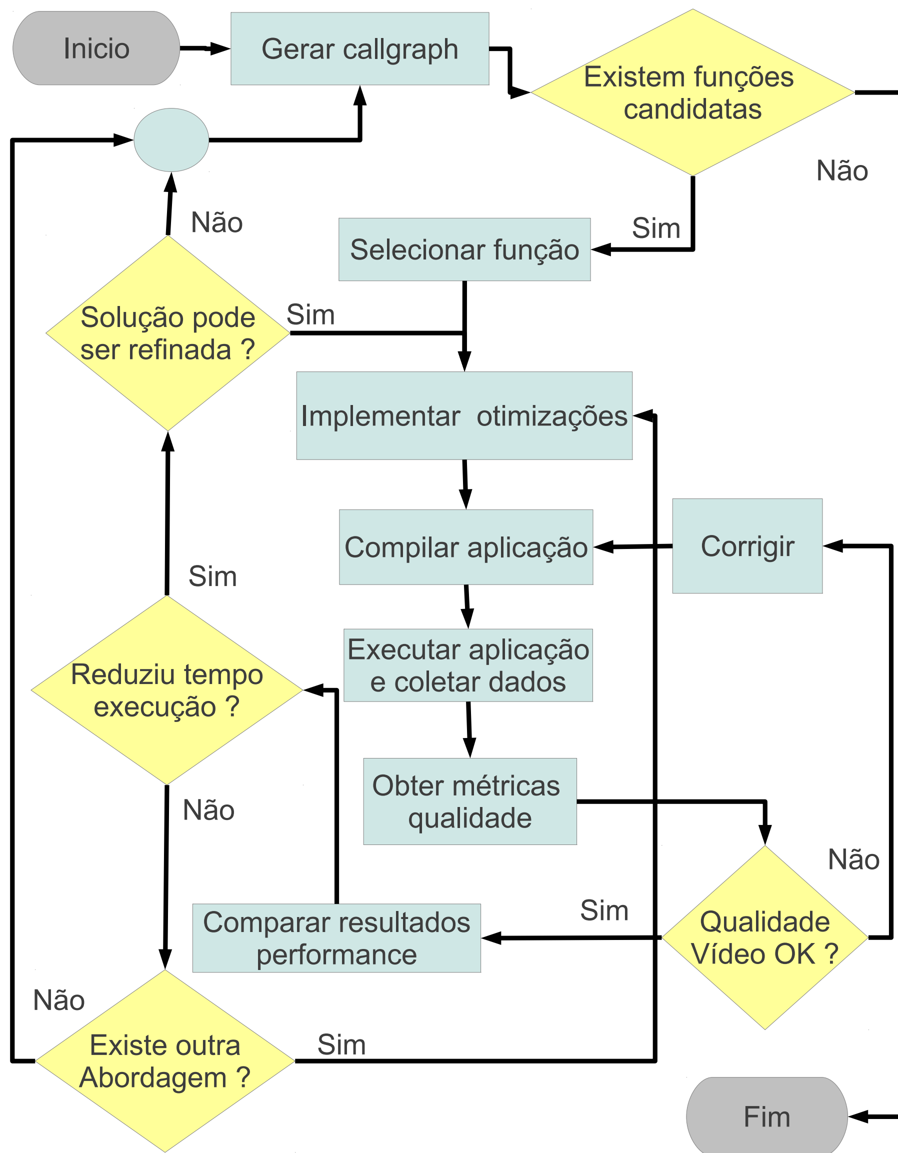
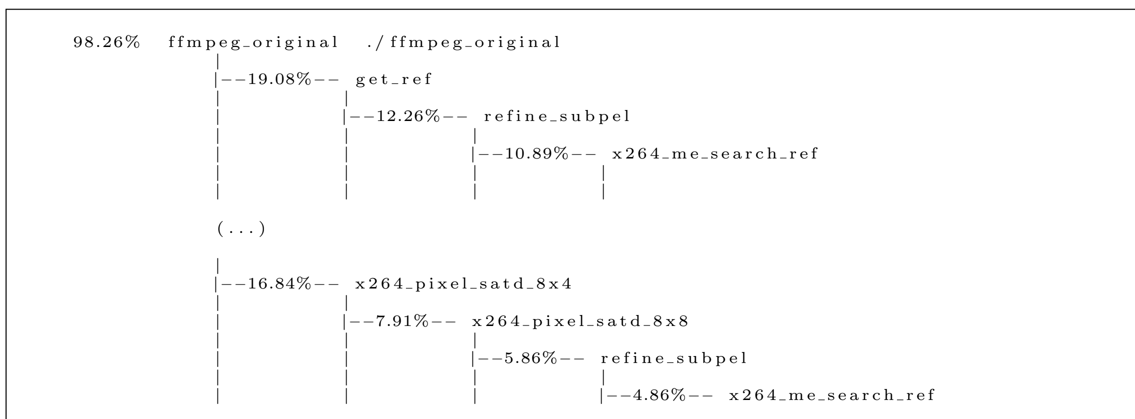


Figura 29: Fluxograma da análise e implementação
Fonte: Autoria própria

```
----------------------------------------------
 Percent |   Source code & Disassembly of ffmpeg_original
----------------------------------------------
         :
         :
         :
         :   Disassembly of section .text:
         :
         :   080f86e5 <x264_pixel_satd_8x4>:
         :       }
         :       return sum >> 1;
         :   }
         :
         :   static NOINLINE int x264_pixel_satd_8x4( pixel *pix1, int i_pix1, pixel *pix2, int i_pix2 )
         :   {
    0.86 :    80f86e5:    55                        push   %ebp
    0.05 :    80f86e6:    31 c9                     xor    %ecx,%ecx
    0.00 :    80f86e8:    89 e5                     mov    %esp,%ebp
    0.49 :    80f86ea:    57                        push   %edi
    0.02 :    80f86eb:    56                        push   %esi
    0.06 :    80f86ec:    53                        push   %ebx
    0.06 :    80f86ed:    83 ec 54                  sub    $0x54,%esp
    0.45 :    80f86f0:    8b 55 08                  mov    0x8(%ebp),%edx
    1.56 :    80f86f3:    8b 45 10                  mov    0x10(%ebp),%eax
         :       uint32_t tmp[4][4];
         :       uint32_t a0, a1, a2, a3;
         :       int sum = 0;
         :       for( int i = 0; i < 4; i++, pix1 += i_pix1, pix2 += i_pix2 )
         :       {
         :           a0 = (pix1[0] - pix2[0]) + ((pix1[4] - pix2[4]) << 16);
    1.49 :    80f86f6:    0f b6 18                  movzbl (%eax),%ebx
    3.11 :    80f86f9:    0f b6 32                  movzbl (%edx),%esi
   0.33 :   80f86fc:    0f b6 78 04              movzbl 0x4(%eax),%edi
    1.42 :    80f8700:    29 de                     sub    %ebx,%esi
    0.06 :    80f8702:    0f b6 5a 04               movzbl 0x4(%edx),%ebx
    1.02 :    80f8706:    29 fb                     sub    %edi,%ebx
    1.58 :    80f8708:    c1 e3 10                  shl    $0x10,%ebx
    1.49 :    80f870b:    8d 1c 1e                  lea    (%esi,%ebx,1),%ebx
    1.57 :    80f870e:    89 5d ac                  mov    %ebx,-0x54(%ebp)
         :           a1 = (pix1[1] - pix2[1]) + ((pix1[5] - pix2[5]) << 16);
    1.79 :    80f8711:    0f b6 58 01               movzbl 0x1(%eax),%ebx
    0.00 :    80f8715:    0f b6 72 01               movzbl 0x1(%edx),%esi
    0.01 :    80f8719:    0f b6 78 05               movzbl 0x5(%eax),%edi
    0.02 :    80f871d:    29 de                     sub    %ebx,%esi
    1.82 :    80f871f:    0f b6 5a 05               movzbl 0x5(%edx),%ebx
    0.03 :    80f8723:    29 fb                     sub    %edi,%ebx
    0.00 :    80f8725:    c1 e3 10                  shl    $0x10,%ebx
    1.31 :    80f8728:    8d 1c 1e                  lea    (%esi,%ebx,1),%ebx
    1.72 :    80f872b:    89 5d a8                  mov    %ebx,-0x58(%ebp)

(...)
```

Figura 30: Exemplo de execução do *perf-annotate* com o x264
Fonte: Autoria própria



```
98.26%  ffmpeg_original  ./ffmpeg_original
            |
            |--19.08%-- get_ref
            |          |
            |          |--12.26%-- refine_subpel
            |          |          |
            |          |          |--10.89%-- x264_me_search_ref
            |          |          |          |
            |          |          |          |

            (...)

            |
            |--16.84%-- x264_pixel_satd_8x4
            |          |
            |          |--7.91%-- x264_pixel_satd_8x8
            |          |          |
            |          |          |--5.86%-- refine_subpel
            |          |          |          |
            |          |          |          |--4.86%-- x264_me_search_ref
```

Figura 31: Exemplo de execução do *perf-report –call-graph* com o x.264
Fonte: Autoria própria

```
Performance counter stats for './ffmpeg_modificado (...)' (5 runs):

        8586  context-switches          #      0.000 M/sec   ( +-  21.417% )
 28036217806  cycles                    #      0.000 M/sec   ( +-   0.061% )
 25705074110  bus-cycles                #      0.000 M/sec   ( +-   0.089% )

 12.676355899  seconds time elapsed   ( +-   0.040% )
```

Figura 32: Exemplo de execução do *perf-stat* com o x264
Fonte: Autoria própria

implementação foram criadas as diretivas de compilação indicadas na tabela 18. Uma diretiva adicional, chamada de OPT_BRANCH_PREDICTION, foi criada para controlar a ativação das implementações específicas para a redução dos eventos de *branch-miss*.

Tabela 18: Diretivas de compilação para controle das implementações

Diretiva	Funções e implementações habilitadas
OPT_MC_CHROMA	função mc_chroma
OPT_PIXEL_AVG	função pixel_avg
OPT_SAD	funções para SAD
OPT_SATD	implementações gerais para as funções de SATD
OPT_SATD_INTRA	funções SATD intra
OPT_SATD_4x4	funções para SATD 4x4
OPT_BRANCH_PREDICTION	redução de *branch-prediction*

Fonte: Autoria própria

A figura 33 apresenta a implementação da função x264_pixel_satd_8x4, utilizada para calcular a SATD para macroblocos com tamanho 8x4. A versão otimizada dessa função é apresentada na figura 34. Todas as técnicas utilizadas estão aplicadas e a ativação de cada uma delas é feita através das diretivas de compilação da tabela 18. Existem outras funções para SATD e SAD, para os diferentes tamanhos de macroblocos. Elas apresentam uma grande semelhança entre si, diferindo basicamente na quantidade de iterações executadas dentre de cada laço, e as otimizações foram aplicadas sobre todas essas funções.

Inicialmente utilizou-se o tipo de dado apresentado na figura 28 para executar as operações dentro do laço iniciado na linha 6 da figura 33 e que calcula o valor das variáveis a0, a1, a2 e a3. Essa primeira abordagem permitiu reduzir a quantidade de ciclos gastos para o acesso à memória. Na sequência, utilizou-se a técnica de *loop unrolling* (ativada com a diretiva OPT_BRANCH_PREDICTION) para esse mesmo laço, reduzindo-se com isso a quantidade de eventos de *branch-miss* que consequentemente reduziu a quantidade de ciclos registrados para o acesso à memória. Os resultados das duas técnicas aplicadas em conjunto são apresentados nas tabela 20, 21 e 22.

```
 1 static NOINLINE int x264_pixel_satd_8x4( pixel *pix1, int i_pix1, pixel *pix2, int i_pix2 )
 2 {
 3     uint32_t tmp[4][4];
 4     uint32_t a0, a1, a2, a3;
 5     int sum = 0;
 6     for( int i = 0; i < 4; i++, pix1 += i_pix1, pix2 += i_pix2 )
 7     {
 8         a0 = (pix1[0] - pix2[0]) + ((pix1[4] - pix2[4]) << 16);
 9         a1 = (pix1[1] - pix2[1]) + ((pix1[5] - pix2[5]) << 16);
10         a2 = (pix1[2] - pix2[2]) + ((pix1[6] - pix2[6]) << 16);
11         a3 = (pix1[3] - pix2[3]) + ((pix1[7] - pix2[7]) << 16);
12         HADAMARD4( tmp[i][0], tmp[i][1], tmp[i][2], tmp[i][3], a0,a1,a2,a3 );
13     }
14     for( int i = 0; i < 4; i++ )
15     {
16         HADAMARD4( a0, a1, a2, a3, tmp[0][i], tmp[1][i], tmp[2][i], tmp[3][i] );
17         sum += abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
18     }
19     return (((uint16_t)sum) + ((uint32_t)sum>>16)) >> 1;
20 }
```

Figura 33: Versão original SATD 8x4
Fonte: Autoria própria

```
 1 #ifdef OPT_SATD
 2 static NOINLINE int optim_x264_pixel_satd_8x4( pixel *pix1, int i_pix1, pixel *pix2, int i_pix2 )
 3 {
 4         pixel_64_t u64_pix1;
 5         pixel_64_t u64_pix2;
 6     uint32_t tmp[4][4];
 7     uint32_t a0, a1, a2, a3;
 8         int sum = 0;
 9 
10 #ifdef OPT_BRANCH_PREDICTION
11 #define OPT_BP_SATD_8x4_A(idx) \
12                 u64_pix1.u64 = (uint64_t *)pix1;        \
13                 u64_pix2.u64 = (uint64_t *)pix2;        \
14         a0 = (u64_pix1.u8[0] - u64_pix2.u8[0]) + ((u64_pix1.u8[4] - u64_pix2.u8[4]) << 16);     \
15         a1 = (u64_pix1.u8[1] - u64_pix2.u8[1]) + ((u64_pix1.u8[5] - u64_pix2.u8[5]) << 16);     \
16         a2 = (u64_pix1.u8[2] - u64_pix2.u8[2]) + ((u64_pix1.u8[6] - u64_pix2.u8[6]) << 16);     \
17         a3 = (u64_pix1.u8[3] - u64_pix2.u8[3]) + ((u64_pix1.u8[7] - u64_pix2.u8[7]) << 16);     \
18                 pix1 += i_pix1; \
19                 pix2 += i_pix2; \
20         HADAMARD4( tmp[idx][0], tmp[idx][1], tmp[idx][2], tmp[idx][3], a0,a1,a2,a3 );
21 
22                 OPT_BP_SATD_8x4_A(0);
23                 OPT_BP_SATD_8x4_A(1);
24                 OPT_BP_SATD_8x4_A(2);
25                 OPT_BP_SATD_8x4_A(3);
26 
27 #else /* OPT_BRANCH_PREDICTION */
28     for( int i = 0; i < 4; i++ )
29     {
30                 u64_pix1.u64 = (uint64_t *)pix1;
31                 u64_pix2.u64 = (uint64_t *)pix2;
32 
33         a0 = (u64_pix1.u8[0] - u64_pix2.u8[0]) + ((u64_pix1.u8[4] - u64_pix2.u8[4]) << 16);
34         a1 = (u64_pix1.u8[1] - u64_pix2.u8[1]) + ((u64_pix1.u8[5] - u64_pix2.u8[5]) << 16);
35         a2 = (u64_pix1.u8[2] - u64_pix2.u8[2]) + ((u64_pix1.u8[6] - u64_pix2.u8[6]) << 16);
36         a3 = (u64_pix1.u8[3] - u64_pix2.u8[3]) + ((u64_pix1.u8[7] - u64_pix2.u8[7]) << 16);
37                 pix1 += i_pix1;
38                 pix2 += i_pix2;
39 
40         HADAMARD4( tmp[i][0], tmp[i][1], tmp[i][2], tmp[i][3], a0,a1,a2,a3 );
41     }
42 #endif /* OPT_BRANCH_PREDICTION */
43 #ifdef OPT_BRANCH_PREDICTION
44 #define OPT_BP_SATD_8x4_B(idx) \
45         HADAMARD4( a0, a1, a2, a3, tmp[0][idx], tmp[1][idx], tmp[2][idx], tmp[3][idx] );      \
46         sum += abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
47 
48         OPT_BP_SATD_8x4_B(0);
49         OPT_BP_SATD_8x4_B(1);
50         OPT_BP_SATD_8x4_B(2);
51         OPT_BP_SATD_8x4_B(3);
52 
53 #else /* OPT_BRANCH_PREDICTION */
54     for( int i = 0; i < 4; i++ )
55     {
56         HADAMARD4( a0, a1, a2, a3, tmp[0][i], tmp[1][i], tmp[2][i], tmp[3][i] );
57         sum += abs2(a0) + abs2(a1) + abs2(a2) + abs2(a3);
58     }
59 #endif /* OPT_BRANCH_PREDICTION */
60     return (((uint16_t)sum) + ((uint32_t)sum>>16)) >> 1;
61 }
62 #endif /* OPT_SATD */
```

Figura 34: Implementação da otimização para SATD 8x4
Fonte: Autoria própria

O codificador x264 oferece versões otimizadas para as suas principais funções e algumas dessas otimizações utilizam-se de implementações baseadas em instruções SIMD, disponíveis em algumas arquiteturas de *hardware* como por exemplo a x86. Para a utilização dessas otimizações, o suporte ao *assembly* precisa ser habilitado no x264 durante a compilação e um compilador/*assembler* precisa estar instalado no ambiente de desenvolvimento.

As implementações desenvolvidas para este trabalho tiveram como foco a melhora da performance do codificador, preservando a qualidade objetiva do vídeo codificado e mantendo o binário do codificador com um tamanho reduzido. Por esse motivo, o codificador foi compilado com as otimizações para redução do tamanho do binário habilitadas no compilador, e para o caso do compilador gcc foi utilizada a opção -*Os* durante a compilação do x264 e a opção *–enable-small* para a compilação do ffmpeg. A portabilidade da solução entre diferentes arquiteturas de *hardware* também foi considerada e por esse motivo o suporte ao *assembly*, e consequentemente o suporte a SIMD, foi desabilitado nas versões consideradas na avaliação do desempenho da otimização (versões Osc e Oss).

3.3.2 RESULTADOS

Os resultados a seguir apresentam os dados de performance da execução de várias versões do codificador sobre um conjunto de vídeos digitais, permitindo uma comparação do desempenho da solução de otimização adotada, em relação a outras abordagens. Esses resultados foram coletados no equipamento *notebook1*, cujas características estão indicadas na seção 3.1.

Para avaliar a otimização proposta, foram geradas 4 versões diferentes do codificador, sendo que em duas das versões (Osc e O2c) foram ativadas as otimizações propostas e nas outras duas versões (Oss e O2s), manteve-se as otimizações desabilitadas. As demais versões do codificador foram geradas e executadas para permitir uma melhor análise e comparação dos resultados obtidos em relação às otimizações disponíveis no codificador x264. A tabela 19 contêm uma pequena descrição de cada uma das versões, o tamanho do binário produzido e a variação do tamanho em relação à versão Oss.

As tabelas 20, 21 e 22 contém os dados obtidos do relatório da ferramenta *perf-stat* a partir de 5 execuções sucessivas de cada uma das 8 versão do codificador sobre as 9 sequências de vídeo utilizadas. Os dados apresentados nessas tabelas são valores médios calculados pela ferramenta utilizada. A coluna Tempo corresponde ao tempo médio de execução do codificador para uma dada sequência de vídeo. Nas colunas *cycles* e *bus-*

Tabela 19: Versões dos codificadores

Otimização	Opção no compilador	Tamanho do binário [*bytes*]	Variação [%]
O2	nível O2	2164000	67,100
02a	nível O2 com *assembly* habilitado	2684896	107,322
O2c	nível O2 com as otimizações propostas	2164000	67,100
O2s	nível O2 sem as otimizações propostas	2164000	67,100
O3	nível O3	2389280	84,495
O3a	nível O3 com *assembly* habilitado	2910176	124,718
Osc	nível Os com as otimizações propostas	1299132	0,316
Oss	nível Os sem as otimizações propostas	1295036	0

Fonte: Autoria própria

cycles são apresentados os valores médios correspondentes ao número de ciclos totais utilizados pela CPU e a quantidade de ciclos utilizados durante o acesso ao barramento externo respectivamente. A quantidade de desvios condicionais e a quantidade de erros de predição dos desvios para a execução no *pipeline* do processador estão respectivamente nas colunas *branches* e *branch-miss*. As colunas *cache-reference* e *cache-miss* contêm os valores médios da quantidade de referências feitas à memória *cache* do processador e a quantidade de vezes em que o conteúdo de interesse não foi encontrado na memória *cache*, incorrendo em atrasos adicionais devido à necessidade de transferir a informação requisitada da memória externa para a memória *cache*. Os valores para as taxas CPI e BMPI foram obtidos respectivamente com as equações (5) e (6).

As reduções nos valores médios para os eventos reportados nas tabelas 20, 21 e 22 estão indicados na tabela 23. Essas diferenças referem-se à variação nos valores médios dos eventos do codificador **Osc** em relação ao codificador **Oss**.

As métricas objetivas de qualidade do vídeo foram obtidas com a utilização da ferramenta MSU (MSU, 2011). O valor médio para cada uma das métricas é apresentado na tabela 24, bem como o tamanho final dos arquivos codificados.

Tabela 20: Dados coletados durante a execução dos codificadores

Vídeo	**Otimização**	**Tempo**	***cycles***	***bus-cycles***	***branches***	***branch-miss***	***cache-reference***	***cache-miss***	**CPI**	**BMPI**
akiyo	**Oss**	13,156	28.702.317.074	2.613.287.556	5.875.421.586	98.371.769	138.290.017	4.210.035	0,487	0,961
	Osc	10,993	23.967.480.999	2.182.644.689	2.961.110.915	79.928.843	141.086.716	4.522.584	0,480	0,957
	O2	11,566	25.237.815.605	2.298.093.612	4.706.434.102	104.075.407	146.364.541	4.388.304	0,439	0,773
	O2c	10,517	22.921.857.117	2.087.694.785	2.287.252.996	81.329.274	152.343.654	4.812.621	0,447	0,606
	O2s	11,566	25.238.745.505	2.298.324.130	4.711.690.984	104.537.378	146.838.125	4.399.434	0,439	0,775
	O2a	2,973	6.458.316.636	589.580.611	680.753.969	46.293.182	125.882.966	6.860.291	0,741	1,753
	O3	9,948	21.666.893.260	1.973.963.054	1.880.850.963	64.938.931	226.057.618	5.219.868	0,457	0,403
	O3a	2,939	6.370.956.917	581.736.247	621.188.356	32.808.571	150.053.448	7.908.940	0,746	1,626
bridge_close	**Oss**	107,180	234.177.897.696	21.311.312.052	47.038.890.877	808.391.334	1.058.879.690	28.969.845	0,491	0,889
	Osc	91,041	198.801.874.944	18.095.474.109	24.688.353.140	696.978.457	1.066.118.315	32.082.390	0,486	0,900
	O2	94,763	206.933.717.225	18.836.698.598	38.056.829.066	884.068.009	1.131.577.718	30.631.013	0,443	0,723
	O2c	86,953	189.915.632.327	17.283.448.972	19.467.905.904	685.621.791	1.158.896.811	33.515.974	0,452	0,574
	O2s	94,800	207.078.707.468	18.846.136.436	38.045.015.074	887.155.795	1.131.369.445	30.457.789	0,443	0,723
	O2a	25,159	54.844.064.018	4.993.642.952	5.753.886.591	320.064.518	999.222.689	50.888.826	0,724	1,778
	O3	82,385	179.813.774.929	16.369.833.489	16.200.346.753	559.019.733	1.853.621.745	34.284.340	0,463	0,395
	O3a	24,939	54.362.086.714	4.951.046.399	5.322.436.989	278.417.131	1.115.412.145	51.602.297	0,730	1,562
bridge_far	**Oss**	136,828	298.949.311.710	27.207.525.328	61.652.234.007	1.090.879.774	1.418.911.163	30.704.324	0,485	0,896
	Osc	115,215	251.585.028.767	22.897.047.386	31.385.994.637	940.288.218	1.431.758.623	34.213.034	0,480	0,922
	O2	121,129	264.646.119.779	24.082.393.323	49.269.711.932	1.207.494.512	1.527.999.879	33.292.311	0,441	0,737
	O2c	111,116	242.684.072.429	22.088.505.348	24.407.467.758	935.658.501	1.549.437.494	36.152.790	0,450	0,586
	O2s	121,201	264.741.403.677	24.094.070.019	49.260.776.393	1.209.421.088	1.527.001.013	33.013.382	0,441	0,738
	O2a	32,896	70.153.069.327	6.394.965.611	7.553.107.071	433.392.778	1.294.894.008	58.872.744	0,714	2,035
	O3	104,459	228.072.783.636	20.759.951.262	20.410.660.841	771.952.442	2.434.523.629	37.412.001	0,458	0,391
	O3a	31,611	68.926.151.145	6.274.509.896	6.947.337.004	378.768.267	1.423.817.206	59.463.921	0,714	1,812

Fonte: Autoria própria

Tabela 21: Dados coletados durante a execução dos codificadores

Vídeo	**Otimização**	**Tempo**	***cycles***	***bus-cycles***	***branches***	***branch-miss***	***cache-reference***	***cache-miss***	**CPI**	**BMPI**
bus	**Oss**	11,410	24.913.544.311	2.268.297.106	5.402.583.620	83.884.343	121.955.908	2.139.765	0,473	0,768
	Osc	9,377	20.447.892.930	1.862.950.144	2.614.339.539	65.168.942	122.357.302	2.364.745	0,461	0,776
	O2	9,958	21.734.046.272	1.978.242.088	4.210.928.471	90.123.115	126.990.822	2.369.510	0,427	0,679
	O2c	9,134	19.923.963.577	1.815.199.187	1.962.241.999	64.326.525	127.570.169	2.357.899	0,436	0,500
	O2s	9,965	21.746.981.849	1.979.776.391	4.213.312.642	90.152.427	127.314.047	2.358.604	0,427	0,682
	O2a	2,538	5.367.179.929	490.212.132	533.953.697	33.508.909	107.756.687	4.264.567	0,724	2,237
	O3	8,613	18.791.609.071	1.711.423.398	1.812.544.979	56.375.882	187.277.745	2.642.395	0,444	0,346
	O3a	2,466	5.331.339.119	487.204.039	508.863.098	32.022.021	117.904.686	4.236.998	0,727	2,147
foreman	**Oss**	22,836	49.840.109.123	4.537.626.455	10.655.873.482	172.587.214	236.198.073	4.778.371	0,478	0,807
	Osc	18,950	41.367.421.311	3.765.568.993	5.320.491.774	134.620.430	237.000.677	5.011.929	0,469	0,806
	O2	19,953	43.577.005.808	3.965.670.918	8.336.458.016	182.560.330	250.880.625	5.246.385	0,432	0,688
	O2c	18,331	40.021.720.429	3.643.288.596	4.008.676.958	135.099.201	252.568.653	5.304.571	0,441	0,531
	O2s	19,961	43.568.701.344	3.967.152.309	8.334.333.656	183.751.050	251.761.614	5.217.751	0,432	0,710
	O2a	5,080	10.837.924.859	989.272.985	1.078.938.096	70.514.830	210.458.193	8.930.916	0,722	2,331
	O3	17,334	37.842.935.742	3.444.928.861	3.623.794.221	114.600.672	367.163.490	5.474.946	0,451	0,357
	O3a	4,938	10.744.253.647	979.297.037	1.020.240.563	64.200.032	233.014.706	9.128.774	0,728	2,164
highway	**Oss**	167,944	366.900.647.749	33.393.418.032	79.300.108.172	1.221.422.949	1.754.094.953	32.501.447	0,472	0,800
	Osc	138,786	302.878.041.505	27.587.163.895	38.582.978.164	962.415.550	1.772.503.071	35.101.830	0,463	0,802
	O2	147,159	321.466.616.011	29.257.092.152	62.255.981.413	1.307.686.922	1.864.619.398	34.615.426	0,428	0,672
	O2c	134,787	294.451.460.026	26.797.238.168	29.230.790.834	950.059.121	1.880.841.172	36.605.630	0,437	0,515
	O2s	147,167	321.491.194.273	29.260.135.867	62.249.913.169	1.312.445.435	1.868.199.070	34.835.727	0,428	0,678
	O2a	36,798	78.932.225.512	7.191.394.100	8.118.671.453	490.322.162	1.536.051.806	62.915.905	0,706	2,240
	O3	126,797	276.977.701.292	25.207.088.945	26.720.944.221	809.132.446	2.728.505.164	37.900.793	0,443	0,349
	O3a	36,152	78.757.053.924	7.168.697.187	7.673.037.073	461.472.390	1.696.200.634	63.623.706	0,713	2,096

Fonte: Autoria própria

Tabela 22: Dados coletados durante a execução dos codificadores

Vídeo	**Otimização**	**Tempo**	***cycles***	***bus-cycles***	***branches***	***branch-miss***	***cache-reference***	***cache-miss***	**CPI**	**BMPI**
mobile	**Oss**	21,480	46.906.398.150	4.269.342.089	10.107.426.694	172.570.813	218.221.325	4.413.417	0,480	0,862
	Osc	17,796	38.835.904.063	3.536.463.364	5.027.971.817	135.535.923	219.664.483	4.762.721	0,471	0,869
	O2	18,738	40.901.096.066	3.723.589.814	7.887.284.324	183.157.597	230.899.516	4.650.264	0,434	0,715
	O2c	17,229	37.597.670.038	3.423.509.008	3.792.528.191	141.358.058	240.149.935	5.012.646	0,443	0,568
	O2s	18,730	40.871.276.443	3.721.373.719	7.883.999.509	182.046.693	231.048.352	4.816.428	0,434	0,717
	O2a	4,921	10.507.478.113	959.271.223	1.101.959.575	67.890.312	198.188.241	8.744.493	0,719	2,260
	O3	16,254	35.486.121.023	3.230.138.510	3.399.489.734	114.099.741	359.959.528	5.074.558	0,453	0,381
	O3a	4,781	10.390.902.328	947.652.108	1.028.290.556	61.296.660	220.841.586	8.770.101	0,722	2,094
news	**Oss**	15,383	33.565.117.093	3.056.713.823	6.965.568.205	118.633.807	161.695.882	4.505.778	0,483	0,922
	Osc	12,829	27.983.517.624	2.547.854.131	3.451.581.710	93.292.661	162.508.658	4.822.114	0,474	0,898
	O2	13,533	29.534.004.690	2.689.316.885	5.557.481.985	122.721.334	170.240.699	4.654.349	0,436	0,743
	O2c	12,346	26.917.875.171	2.451.634.551	2.679.755.179	94.433.648	175.318.956	5.403.581	0,444	0,565
	O2s	13,529	29.545.103.157	2.689.334.869	5.561.435.100	123.118.543	168.817.305	4.744.954	0,436	0,733
	O2a	3,511	7.353.064.966	671.748.741	735.090.764	44.271.266	147.281.708	7.632.460	0,740	2,050
	O3	11,701	25.514.271.300	2.323.644.758	2.317.636.682	77.114.898	261.655.000	5.301.886	0,454	0,388
	O3a	3,369	7.332.121.692	668.373.116	698.225.131	41.462.618	161.612.420	7.838.918	0,746	1,862
tempete	**Oss**	20,049	43.793.992.674	3.986.234.337	9.387.019.316	159.383.843	209.080.828	4.000.758	0,478	0,818
	Osc	16,607	36.257.389.013	3.300.535.746	4.671.822.930	121.399.191	201.863.139	4.204.330	0,469	0,825
	O2	17,490	38.199.124.640	3.477.046.018	7.313.044.646	165.030.611	214.752.630	4.271.750	0,432	0,690
	O2c	16,098	35.157.670.381	3.199.607.360	3.509.016.151	121.765.107	218.142.672	4.307.429	0,442	0,544
	O2s	17,493	38.203.226.929	3.477.187.655	7.316.262.468	165.654.917	216.002.985	4.199.671	0,432	0,699
	O2a	4,559	9.625.279.862	879.049.479	987.993.037	64.003.893	184.504.064	7.504.842	0,719	2,236
	O3	15,203	33.187.285.138	3.021.280.327	3.171.365.849	103.694.114	326.791.479	4.439.579	0,452	0,377
	O3a	4,385	9.524.509.155	868.582.552	930.591.225	58.786.478	203.701.279	7.560.016	0,723	2,107

Fonte: Autoria própria

Tabela 23: Variação nos eventos para o codificador - Osc x Oss

Vídeo	Variação [%]				
	cycles	*bus-cycles*	*branch-miss*	*cache-miss*	tempo
Akiyo	-16,496	-16,479	-18,748	7,424	-16,441
Bridge close	-15,106	-15,090	-13,782	10,744	-15,058
Bridge far	-15,844	-15,843	-13,805	11,427	-15,796
Bus	-17,925	-17,870	-22,311	10,514	-17,818
Foreman	-17	-17,015	-21,999	4,888	-17,017
Highway	-17,450	-17,387	-21,205	8,001	-17,362
Mobile	-17,206	-17,166	-21,461	7,915	-17,151
News	-16,629	-16,647	-21,361	7,021	-16,603
Tempete	-17,209	-17,202	-23,832	5,088	-17,168

Fonte: Autoria própria

Tabela 24: Características dos vídeos codificados

Vídeo	*assembly*	Métrica			Tamanho
		MSE	PSNR	SSIM	[*bytes*]
Akiyo	Sim	2713,55347	13,79499	0,69191	293092
	Não	2713,37329	13,79527	0,69145	294784
Bridge close	Sim	694,29205	19,71502	0,83840	2401512
	Não	694,40143	19,71433	0,83846	2397000
Bridge far	Sim	606,33160	20,30332	0,85663	2596092
	Não	606,49030	20,30219	0,85670	2596468
Bus	Sim	5130,25049	11,02911	0,18952	160364
	Não				
Foreman	Sim	4121,83887	11,97956	0,41070	359832
	Não	3883,05273	12,23871	0,41385	359456
Highway	Sim	1407,01318	16,64760	0,77570	2405836
	Não	1409,73865	16,63919	0,77555	2398692
Mobile	Sim	4616,57959	11,48729	0,14519	357764
	Não				
News	Sim	4739,47217	11,37320	0,53708	320540
	Não	4719,23877	11,39178	0,53840	319976
Tempete	Sim	5051,85547	11,09602	0,22352	322420
	Não				

Fonte: Autoria própria

3.4 CONCLUSÕES

Nos resultados obtidos nas seções 3.2 e 3.3, é possível perceber uma relação direta entre a redução percentual nos ciclos gastos durante o acesso ao barramento externo e a redução percentual do tempo de execução do codificador. Como neste trabalho, a redução no tempo de execução foi considerado como medida da melhora na performance, podemos concluir que para os casos apresentados, a redução na quantidade de ciclos para o acesso ao barramento externo está diretamente ligada, em termos percentuais, à melhora na performance do codificador.

Os ganhos obtidos na seção 3.2 foram superiores aos da seção 3.3, conforme as tabelas 15, 16 e 23. Uma possível explicação para essa diferença pode estar no fato de que na seção 3.2, foram executadas operações muito simples e que necessitavam de um acesso mais intenso à memória externa pois após a execução das operações, o resultado era imediatamente armazenado e uma nova operação se iniciava. No caso do codificador de vídeo da seção 3.3, operações mais complexas são realizadas antes de se obter o resultado final.

Foram obtidos ganhos de performance ainda mais expressivos com a redução dos eventos de *branch-miss* gerados pelas funções SAD e SATD. Para o codificador com acesso à memória melhorado e redução nos eventos de *branch-miss*, foi possível atingir reduções de até 17,8% no tempo de codificação.

Para a GPU, os resultados apresentados na seção 3.2 foram superiores aos obtidos com acessos de 64 bits, com a desvantagem de ter sido necessário reescrever a aplicação para adequá-la à arquitetura utilizada nesse dispositivo.

A quantidade de eventos de *cache-miss* aumentou na versão Osc, indicando uma degradação no aproveitamento da memória *cache* do sistema, criando uma oportunidade de melhoria na localidade dos dados, podendo assim otimizar ainda mais o desempenho do codificador.

A qualidade objetiva dos vídeos codificados, representada pelas métricas PSNR, MSE e SSIM, apresentou pequenas variações para as versões do codificador com *assembly* habilitado (O2a e O3a), conforme apresentado na tabela 24.

O tamanho do arquivo codificado aumentou para as versões com *assembly* habilitado, e isso gera um valor diferente para o valor médio das métricas calculadas. As implementações em *assembly* usam versões modificadas das funções, ao invés de usar

otimizações internas ao compilador, e isso pode gerar pequenas diferenças durante o cálculo da SAD e SATD, alterando o resultado final da estimação de movimento e gerando pequenas variações no arquivo codificado. Durante a execução das sequências de vídeo codificadas, nenhum problema de qualidade foi percebido.

O desempenho dos codificadores com *assembly* habilitado, e que consequentemente utilizam as instruções SIMD dos processadores Intel como otimização, é superior ao das outras versões geradas sem esses recursos. Essas otimizações não podem ser utilizadas em outras arquiteturas de processadores e por esse motivo, não foram objeto de estudo deste trabalho, sendo que os resultados apresentados foram obtidos apenas para permitir uma comparação entre a otimização proposta e as otimizações disponíveis originalmente no codificador.

A versão do codificador otimizada para redução do tamanho do binário (Osc) e que é o objeto deste estudo, apresenta uma redução de 66,8% e 107% em relação as versões O2 e O2a respectivamente, e uma redução de 84% e 124% no tamanho quando comparada com as versões O3 e O3a respectivamente. Em relação ao desempenho dos codificadores, a versão Osc tem uma performance superior à versão O2 e inferior quando comparada com a versão O3.

O acesso ao barramento externo dos computadores atuais ainda é um gargalo que limita a utilização de todo o potencial dos processadores. As arquiteturas de *hardware* tem evoluído para diminuir essa limitação mas ela estará presente por muito tempo ainda. Além disso, é importante aproveitar de forma eficiente os recursos oferecidos pelos processadores modernos como os *pipelines* e as memórias *cache*. Por esse motivo, o método de analise utilizado, baseado em ferramentas de *software profiling* direcionadas à análise a partir dos contadores de performance, mostrou-se uma forma eficiente e menos intrusiva que a utilização de ferramentas como o gprof, que requer a compilação de uma versão especial da aplicação para tornar possível a coleta de dados da execução.

4 CONCLUSÕES E TRABALHOS FUTUROS

Com base nas informações apresentadas no capítulo 2, é possível perceber a diferença nas taxas de transmissão existentes entre as diversas interfaces de comunicação e o processador. Essa diferença entre a capacidade de processamento das CPUs modernas e a velocidade das interfaces de comunicação tende a diminuir com o passar do tempo, como é o caso da comunicação entre processadores da família Core i7 e as memórias externas, cujo controlador foi integrado ao encapsulamento do processador, permitindo atingir taxas de transferências mais altas que as versões anteriores.

O aumento na taxa de transmissão de dados entre a CPU e os dispositivos periféricos e bancos de memória está diretamente relacionado à integração desses com a arquitetura interna da CPU. Porém os custos de produção aumentam à medida que mais dispositivos são integrados, sendo essa uma das razões para a utilização de memórias *cache*, que apresentam uma taxa de transferência superior e baixa capacidade de armazenamento, com o objetivo de manter uma boa relação custo x benefício.

Por esse motivo, é importante o uso racional e eficiente desses recursos, de modo a minimizar o impacto causado por eles na performance dos sistemas, bem como possibilitar o uso de todo o poder computacional das CPUs modernas, evitando que as mesmas permaneçam "bloqueadas" devido a acessos desnecessários a interfaces de comunicação mais lentas ou desperdicem os recursos de *hardware* como os *pipelines.*

O foco deste trabalho foi a análise do impacto sobre o acesso à memória externa do computador, que são os dispositivos externos com maiores taxas de transferência de dados. A utilização de dispositivos de armazenamento mais lentos como por exemplo os discos rígidos, tendem a causar impactos ainda maiores na performance.

A melhoria no acesso à de memória é de certa forma mais genérica que outras melhorias implementadas, pois pode, em maior ou menor escala, ser aproveitada em diversas arquiteturas de *hardware*, sendo que o principal cuidado que se deve tomar é adequar a implementação para o tamanho do barramento disponível na arquitetura de destino da

aplicação.

Uma outra técnica que tem por objetivo minimizar o acesso à memória externa, é o uso de memórias *cache*, disponíveis tanto em *Desktops*, quanto em processadores para sistemas embarcados. A desvantagem dessa técnica é que ela requer um maior conhecimento da arquitetura do *hardware* de forma a usar de maneira eficiente esse tipo de memória, que é mais rápida porém de tamanho reduzido.

Outra técnica empregada neste trabalho, foi a redução dos eventos de *branch-miss* na CPU, o que indiretamente reduziu os acessos ao barramento externo, além de oferecer uma melhor utilização dos *pipelines*.

O uso dos recursos de *hardware* como os contadores de performance, em conjunto com ferramentas de *software profiling* como o **perf**, mostrou-se bastante eficiente e promissor. Foi possível analisar a aplicação sem a necessidade de alteração do código fonte para a geração de um binário especial, permitindo dessa forma a obtenção de resultados mais reais com relação a execução da aplicação.

Uma das motivações para o inicio desse estudo foi a dificuldade em executar o codificador x264 no sistema de captura de áudio e vídeo, equipado com um processador Blackfin Dual-Core, disponível no laboratório LCD da UTFPR. A partir dos resultados obtidos, um novo estudo poderá ser desenvolvido com o objetivo de adaptar as implementações e métodos para essa arquitetura. A maior dificuldade pode ser a falta de suporte, pelo Linux, às ferramentas *perf* para a arquitetura Blackfin, porém, existem iniciativas na comunidade de desenvolvimento visando preencher essa lacuna.

A utilização de GPUs para a codificação de vídeo, principalmente na execução de funções criticas como as relacionadas ao cálculo dos valores de SAD e SATD também podem ser explorado em trabalhos futuros. Os resultados obtidos com a GPU para este trabalho foram bastante restritos e fazem parte de um pequeno estudo iniciado com o objetivo de mapear os impactos e requisitos necessários para a utilização desse recurso.

Como consequência deste trabalho, o seguintes resultados foram obtidos:

- Definição de um método que permite analisar e melhorar a performance de uma aplicação com o uso de ferramentas de *software profiling* em conjunto com os contadores de performance do *hardware*.
- Melhora na performance do codificador x264, permitindo que no futuro, novos trabalhos sejam desenvolvidos, inclusive com foco em sistemas embarcados.

REFERÊNCIAS

ABDELAZIMA, A.; VARLEYA, M.; AIT-BOUDAOUDB, D. Effect of the hadamard transform on motion estimation of different layers in video coding. In: . [s.n.], 2010. XXXVIII, p. 1–4. Acessado em Agosto de 2011. Disponível em: <http://www.isprs.org/proceedings/XXXVIII/part5/papers/70.pdf>.

ALBINI, F. L. P. **Geração de Artefatos em Vídeo Digital**. Dissertação (Mestrado) — Universidade Tecnológica Federal do Paraná, 2009.

ASU. **Vídeo Traces**. 2011. Xiph.org Test Media. Acessado em Junho de 2011. Disponível em: <http://trace.eas.asu.edu/yuv/>.

BREWER, J.; SEKEL, J. **PCI Express Technology**. [S.l.], 2004. Acessado em Agosto de 2011. Disponível em: <http://content.dell.com/us/en/corp/d/businesss̃olutionsw̃hitepapersẽn/Documentsw̃p-2004_pciexpress.pdf.aspx?redirect=1>.

COPPEN, S. W. **Optimizing C/C++ with Inline Assembly Programming**. [S.l.], June 2005. Acessado em Agosto de 2011. Disponível em: <http://www.drdobbs.com/184401967;jsessionid=ERKK20AKHZYA HQE1GHOSKHWATMY32JVN>.

DENOLF, K. et al. Memory centric design of an mpeg-4 video encoder. **Circuits and Systems for Video Technology, IEEE Transactions on**, v. 15, n. 5, p. 609–619, May 2005. ISSN 1051-8215.

FFMPEG. **FFmpeg - solution to record, convert and stream audio and video**. 2011. Acessado em Junho de 2011. Disponível em: <http://www.ffmpeg.org>.

FFMPEG. **FFMPEG Git Repository**. 2011. Acessado em Junho 2011. Disponível em: <git://git.ffmpeg.org/ffmpeg>.

FREEDESKTOP. **Video Acceleration API - VA-API**. 2011. Acessado em Agosto de 2011. Disponível em: <http://www.freedesktop.org/wiki/Software/vaapi>.

GAMBLIN, T. **Scalable Performance Measurement and Analysis**. Tese (Doutorado) — University of North Carolina, 2009.

GCC. **Auto-vectorization in GCC**. 2011. Acessado em Agosto de 2011. Disponível em: <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>.

GCC. **GCC - Gnu Compiler**. 2011. Acessado em Junho 2011. Disponível em: <http://gcc.gnu.org>.

GNU. **GNU Home**. 2011. Acessado em Junho 2011. Disponível em: <http://www.gnu.org>.

GPROF. **Gprof**. 2011. Acessado em Junho 2011. Disponível em: <http://www.gnu.org/software/binutils/>.

GPROF. **Gprof Manual**. [S.l.], 2011. Acessado em Agosto 2011. Disponível em: <http://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_1.html#SEC1>.

GUNEY, M. **H.264 Baseline Video Encoder Implementation and Optimization on TMS320DM642 Digital Signal Processor**. Dissertação (Mestrado) — School of Engineering and Natural Sciences of Sabanci University, 2006.

HEIL, T. **PCI Express & Host Based RAID**. 2004. Acessado em Agosto de 2011. Disponível em: <http://www.pcisig.com/developers/main/training_materials/get_document?doc_id=b48e9d7dfce58f16a084b3f51d4d503c80922dad>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture - A Quantitative Approach**. [S.l.]: Morgan Kaufmann, 2006.

INTEL. **Intel 64 and IA-32 Architectures Optimization Reference Manual**. [S.l.], 2009. Acessado em Junho de 2011. Disponível em: <http://www.intel.com/Assets/PDF/manual/248966.pdf>.

INTEL. **Detecting Memory Bandwidth Saturation in Threaded Applications**. Março 2010. Acessado em Junho de 2011. Disponível em: <http://software.intel.com/en-us/articles/detecting-memory-bandwidth-saturation-in-threaded-applications/>.

INTEL. **Intel Corporation**. [S.l.], 2011. Acessado em Agosto de 2011. Disponível em: <http://www.intel.com>.

INTEL. **Intel® 64 and IA-32 Architectures Developer's Manual: Combined Vols. 1, 2, and 3**. 2011. Acessado em Agosto de 2011. Disponível em: <http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-manual-325462.pdf>.

INTEL. **Intel® Integrated Performance Primitives**. 2011. Acessado em Agosto de 2011. Disponível em: <http://software.intel.com/en-us/articles/intel-ipp/#support>.

ITU. **Recommendation ITU-T P.910: Subjective Video Quality Assessment Methods for Multimedia Applications**. 2008.

ITU. **Recommendation ITU-R BT.500-11: Methodology for the Subjective Assessment of the Quality of Television Pictures**. 2009.

KERNEL.ORG. **Perf - The Linux Performance Counter subsystem**. 2011. Acessado em Junho 2011. Disponível em: <https://perf.wiki.kernel.org>.

KHRONOS. **OpenMAX - The Standard for Media Library Portability**. 2011. Acessado em Agosto de 2011. Disponível em: <http://www.khronos.org/openmax/>.

LEE, C. **Board Design Guidelines for PCI ExpressTM Architecture**. 2004. Acessado em Agosto de 2011. Disponível em: <http://www.pcisig.com/developers/main/training_materials/get_document?doc_id=c48e4d9b1409c7f697669d476995348cf1cd183>.

LIN, Y.; LEE, Y.-M.; WU, C.-D. Efficient algorithm for h.264/avc intra frame video coding. **Circuits and Systems for Video Technology, IEEE Transactions on**, v. 20, n. 10, p. 1367 –1372, oct. 2010. ISSN 1051-8215. IEEE file 5580027.

MELO, A. C. d. **The New Linux 'perf' Tools**. 2011. Acessado em Agosto de 2011. Disponível em: <http://vger.kernel.org/ acme/perf/lk2010-perf-paper.pdf>.

MSU. **MSU Quality Measurement Tool**. 2011. Acessado em Agosto dee 2011. Disponível em: <http://compression.ru/video/qualitymeasure/videomeasurementtoolen.html>.

NAISHLOS, D. **Autovectorization in GCC**. 2004. Acessado em Agosto de 2011. Disponível em: <ftp://gcc.gnu.org/pub/gcc/summit/2004/Autovectorization.pdf>.

NASIM S. MASUD, N. K. F.; VIRK, K.; FARRUKH, A. Architectural optimizations for software-based mpeg4 video encoder. In: **EUSIPCO 2005 - 13th European Signal Processing Conference http://www.eurasip.org/Proceedings/Eusipco/Eusipco2005/**. [s.n.], 2005. Acessado em Agosto de 2011. Disponível em: <http://www.eurasip.org/Proceedings/Eusipco/Eusipco2005/defevent/papers/cr1097.pdf>.

NVI. **CUDA GPUs**. 2011. Acessado em Agosto de 2011. Disponível em: <http://developer.nvidia.com/cuda-gpus>.

NVIDIA. **NVIDIA GeForce® GTX 200 GPU Architectural Overview**. 2008. Acessado em Agosto de 2011. Disponível em: <http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf>.

NVIDIA. **CUDA GPU Occupancy Calculator version 2.4**. 2011. Acessado em Agosto de 2011. Disponível em: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/tools/CUDA_Occupancy_Calculator.xls>.

OSTERMANN, J. et al. Video coding with h.264/avc: tools, performance, and complexity. **Circuits and Systems Magazine, IEEE**, v. 4, n. 1, p. 7 – 28, first 2004. ISSN 1531-636X.

PAPI, P. A. P. I. **PAPI - Performance Application Programming Interface**. 2011. Acessado em Agosto de 2011. Disponível em: <http://icl.cs.utk.edu/papi/>.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)**. 4th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123744938, 9780123744937.

PCI-SIG. **PCI Express® 3.0 Frequently Asked Questions**. 2011. Acessado em Agosto de 2011. Disponível em: <http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ3>.

PCI-SIG. **PCI-SIG**. 2011. Acessado em Agosto 2011. Disponível em: <http://www.pcisig.com>.

PHARR, M.; FERNANDO, R. **GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)**. [S.l.]: Addison-Wesley Professional, 2005. ISBN 0321335597.

SANDEEP, S. **GCC-Inline-Assembly-HOWTO**. 2003. Acessado em Agosto de 2011. Disponível em: <http://ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>.

SHEIKH, H.; BOVIK, A. Image information and visual quality. **Image Processing, IEEE Transactions on**, v. 15, n. 2, p. 430 –444, feb. 2006. ISSN 1057-7149.

SHEIKH, H. R. et al. Optimization of a baseline h.263 video encoder on the tms320c6000. 2008. Acessado em Agosto de 2011. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.76.3421>.

SILVA, E. S. R. da. **Especificação, Projeto e Desenvolvimento de Ferramentas de Auxílio à Avaliação Subjetiva de Vídeo**. Dissertação (Mestrado) — Universidade Tecnológica Federal do Paraná, 2009.

SOLOMON, R. **PCI-SIG® Architecture Overview**. 2008. Acessado em Agosto de 2011. Disponível em: <http://www.pcisig.com/developers/main/training_materials/get_document?doc_id=1c15f1fd6a01d53e66f46611684f81ebc722876f>.

SULLIVAN, G. J.; TOPIWALA, P.; LUTHRA, A. The h.264/avc advanced video coding standard: Overview and introduction to the fidelity range extensions. In: **SPIE conference on Applications of Digital Image Processing XXVII**. [S.l.: s.n.], 2004. p. 454–474.

TANENBAUM, A. S. **Organização Estruturada de Computadores**. 5a edição. ed. [S.l.]: Pearson Education do Brasil, 2007.

TROCKI, K. Performance aspects of using various techniques of programming simd extensions of modern general-purpose processors. In: **Information Technology, 2008. IT 2008. 1st International Conference on**. [S.l.: s.n.], 2008. p. 1 –4.

WANG, Z. et al. Image quality assessment: from error visibility to structural similarity. **Image Processing, IEEE Transactions on**, v. 13, n. 4, p. 600 –612, april 2004. ISSN 1057-7149.

WIEGAND, T. et al. Rate-constrained coder control and comparison of video coding standards. **Circuits and Systems for Video Technology, IEEE Transactions on**, v. 13, n. 7, p. 688 – 703, july 2003. ISSN 1051-8215. File IEEE 1218200.

WIEGAND, T. et al. Rate-constrained coder control and comparison of video coding standards. **Circuits and Systems for Video Technology, IEEE Transactions on**, v. 13, n. 7, p. 688 – 703, july 2003. ISSN 1051-8215.

WIEGAND, T. et al. Overview of the h.264/avc video coding standard. **Circuits and Systems for Video Technology, IEEE Transactions on**, v. 13, n. 7, p. 560–576, July 2003. ISSN 1051-8215.

X264. **x264 - a free h264/avc encoder**. 2011. Acessado em Junho 2011. Disponível em: <http://www.videolan.org/developers/x264.html>.

X264. **X264 Git Repository**. 2011. Acessado em Junho 2011. Disponível em: <git://git.videolan.org/x264.git>.