UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RICARDO KINTSCHNER

Arquitetura Cell Broadband Engine Aplicada a Estimativa de Movimento

Trabalho de Diplomação.

Prof. Dr. Alexandre Silva Carissimi
Orientador

Prof. Ronaldo Hüsemann
Co-orientador

Porto Alegre, junho de 2010.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do ECP: Prof. Gilson Inácio Wirth
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

LISTA DE ABREVIATURAS E SIGLAS

CBE	Cell Broadband Engine
CBEA	Cell Broadband Engine Architecure
CIF	Common Intermediate Format
DDR2	Double Data Rate 2
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECC	Error Correcting Code
EIB	Element Interconnection Bus
FIFO	First In, First Out
FPSCR	Floating-Point Status and Control Register
FXU	Fixed-Point Unit
HBR	Hint for Branch
HD	High Definition
I/O	Input/Output
ITU	International Telecommunication Union
IU	Instruction Unit
JM	JVT Model
JPEG	Joint Photographic Experts Group
JVT	Joint Video Team
LS	Local Store
LSU	Load and Store Unit
MAD	Mean Absolute Difference
MFC	Memory Flow Controller
MMIO	Memory-Mapped I/O
MMU	Memory Management Unit
MPC	Multi-Processing Chip
MPEG	Moving Picture Experts Group

PC	Program Counter
PPE	PowerPC Processor Element
PPSS	Power Processor Storage Subsystem
PPU	PowerPC Processor Unit
PS3	PlayStation 3
PSNR	Peak Signal-to-Noise Ratio
RHEL	Red Hat Enterprise Linux
RISC	Reduced Instruction Set Computer
SAD	Sum of Absolute Differences
SCN	SPU Control Unit
SD	Standard Definition
SDK	System Development Kit
SFP	SPU Floating-Point Unit
SFS	SPU Odd Fixed-Point Unit
SFX	SPU Even Fixed-Point Unit
SIMD	Single Instruction, Multiple Data
SLS	SPU Load and Store Unit
SMP	Symmetric Multi-Processing
SPE	Synergistic Processor Element
SPU	Synergistic Processor Unit
SRF	SPU Register File Unit
SSC	SPU Channel and DMA Unit
SSH	Secure Shell
SXU	Synergistic Execution Unit
VSU	Vector/Scalar Unit
VXU	Vector/SIMD Multimedia Extension Unit
XDR	Extreme Data Rate

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este projeto propõe explorar os recursos da arquitetura do processador *Cell Broadband Engine*, visando principalmente à utilização deste em aplicações que exijam elevado grau de paralelismo, tais como algoritmos de codificação de áudio/vídeo. O Cell é um multiprocessador heterogêneo, composto por um processador de uso geral e oito co-processadores vetoriais. Foi desenvolvido em conjunto por três empresas, com o objetivo de acelerar aplicações multimídia e de processamento vetorial. Como estudo de caso da proposta deve-se portar para a arquitetura Cell um algoritmo relevante em modernas aplicações de multimídia: estimativa de movimento na codificação de vídeos digitais.

Palavras-Chave: *Cell Broadband Engine*, processamento paralelo, codificação de vídeo, estimativa de movimento.

ABSTRACT

This project proposes to exploit the resources of the Cell Broadband Engine processor, aiming primarily to its use in applications which need a high level of parallelism, such as audio/video encoding algorithms. The Cell Broadband Engine is a heterogeneous multiprocessor which consists of a general-purpose processor and eight vector/SIMD coprocessors. It was designed jointly by three companies to accelerate multimedia and vector processing applications. The proposed case study is to port to the Cell architecture to a relevant algorithm in modern multimedia applications: motion estimation in video encoding.

Keywords: Cell Broadband Engine, parallel processing, video encoding, motion estimation.

SUMÁRIO

1 INTRODUÇÃO

O aumento da demanda de aplicações multimídia faz com que engenheiros e programadores explorem o uso de arquiteturas dedicadas para DSP (*Digital Signal Processing)* e de extensões SIMD (*Single Instruction Multiple Data*) a fim de agilizar os algoritmos, otimizando o desempenho geral do sistema. Além disso, podem-se explorar também arquiteturas do tipo MPC (*Multi-Processing Chip*) para algoritmos que suportem operações paralelas (GOREN, 2003).

Um exemplo atual de solução que consegue explorar em uma mesma arquitetura praticamente todas essas tecnologias (DSP, SIMD e MPC) é o processador *Cell Broadband Engine* (CBE), desenvolvido em conjunto pelas empresas IBM, Sony e Toshiba. O processador CBE ficou muito conhecido por ser utilizado como plataforma tecnológica para o console de jogos PlayStation 3, da Sony, que pode ser considerado sua primeira grande aplicação comercial (GREHS, 2009).

Esse processador possui uma poderosa arquitetura de multiprocessamento heterogêneo (distintas estruturas computacionais inclusas em uma mesma arquitetura). Um alto grau de paralelismo pode ser obtido pelo CBE através sua arquitetura de nove núcleos de processamento. A arquitetura do CBE é composta por um microprocessador central, denominado *PowerPC Processor Element* (PPE), que é dedicado para operações de controle e interface, e oito processadores de DSP, denominados *Synergistic Processor Elements* (SPEs), que são voltados para processamento vetorial.

De forma geral, a arquitetura inovadora do processador CBE pode ser utilizada para uma grande variedade de aplicações. Deve-se, entretanto, considerar que as aplicações capazes de explorar suas características de maneira mais eficiente são aquelas que trabalham com operações matemáticas (somas, subtrações, multiplicações, rotações entre outras) sobre grandes volumes de dados, como, por exemplo, aplicações de multimídia e de processamento vetorial.

Dentre estas áreas, pode-se destacar atualmente como um campo emergente para aproveitar uma arquitetura como a do CBE os algoritmos de codificação de vídeo. Isso se justifica, pois vídeos digitais possuem tipicamente grandes quantidades de informação (conjuntos de pixels que compõem as sequências de imagens). Os algoritmos de codificação de vídeo, a fim de tornarem viáveis o armazenamento e, principalmente, a transmissão dessas sequências, em tempo-real, utilizam-se de métodos de compressão com custo computacional demasiadamente elevado, principalmente quando se opera com imagens de alta definição (JACK, 2007).

Dentro de um codificador de vídeo, um dos módulos mais relevantes e que demanda grande parte desse custo de processamento é o algoritmo de estimativa de movimento (HÜSEMANN, 2008). O algoritmo de estimativa de movimento tenta reduzir a

redundância temporal nas sequências de vídeo compensando os movimentos realizados durante a sucessão de quadros. As diversas etapas de cálculo envolvidas nesse algoritmo podem ser consideradas como operações independentes, quando se considera as análises que são feitas em regiões geometricamente distintas da imagem, o que levanta a possibilidade de realizá-las de forma independente e paralela.

1.1 Objetivos

Neste trabalho se realiza um estudo a respeito do processador CBE, a fim de se produzir uma solução paralela de um algoritmo de estimativa de movimento de vídeos, buscando-se aumentar o desempenho do algoritmo a partir da exploração das vantagens da arquitetura CBE. Como forma de verificação dos ganhos da proposta, por fim se apresenta uma análise comparativa de desempenho entre a versão básica (não paralela) de um algoritmo de estimativa de movimento e sua versão paralelizada, obtida como resultado deste trabalho, ambas rodando sobre uma plataforma real que emprega o processador CBE.

1.2 Organização do texto

O capítulo 2 apresenta uma descrição teórica da arquitetura CBE, explicando em detalhes os elementos mais importantes, como os diferentes núcleos processadores e o barramento central de transferência de dados. O capítulo 3 explica, de forma geral, alguns conceitos próprios de algoritmos de estimativa de movimento em vídeos, os quais são necessários para o adequado entendimento da aplicação e do algoritmo escolhido. O capítulo 4 descreve o algoritmo proposto propriamente dito, bem como apresenta detalhes de sua implementação. O capítulo 5 descreve os experimentos efetuados, trazendo os resultados práticos obtidos e a análise dos mesmos. Finalmente, o capítulo 6 apresenta as conclusões do trabalho.

2 CELL BROADBAND ENGINE

A *Cell Broadband Engine Architecture* (CBEA) é uma arquitetura desenvolvida em conjunto pelas empresas multinacionais Sony, Toshiba e IBM cujo projeto foi iniciado em 2001 e concluído em 2005, gerando o *Cell Broadband Engine*. Essa solução estende a arquitetura PowerPC de 64 bits da IBM, utilizando uma arquitetura de computação celular, ou seja, uma arquitetura para processamento paralelo onde cada unidade é um processador poderoso (AMORIM, 2005).

O projeto enfatizou um alto grau de paralelismo, dando origem a uma inédita solução multiprocessada em um único chip (MPC) com nove processadores independentes operando sobre uma memória de sistema compartilhada e coerente. A idéia principal foi utilizar um processador central PowerPC, denominado PPE, com funções comuns de controle e com capacidade de executar um sistema operacional, para coordenar oito processadores vetoriais chamados SPEs, estes tendo sido otimizados para processamento intensivo de dados. Uma visão geral da arquitetura CBEA pode ser vista na figura 2.1.

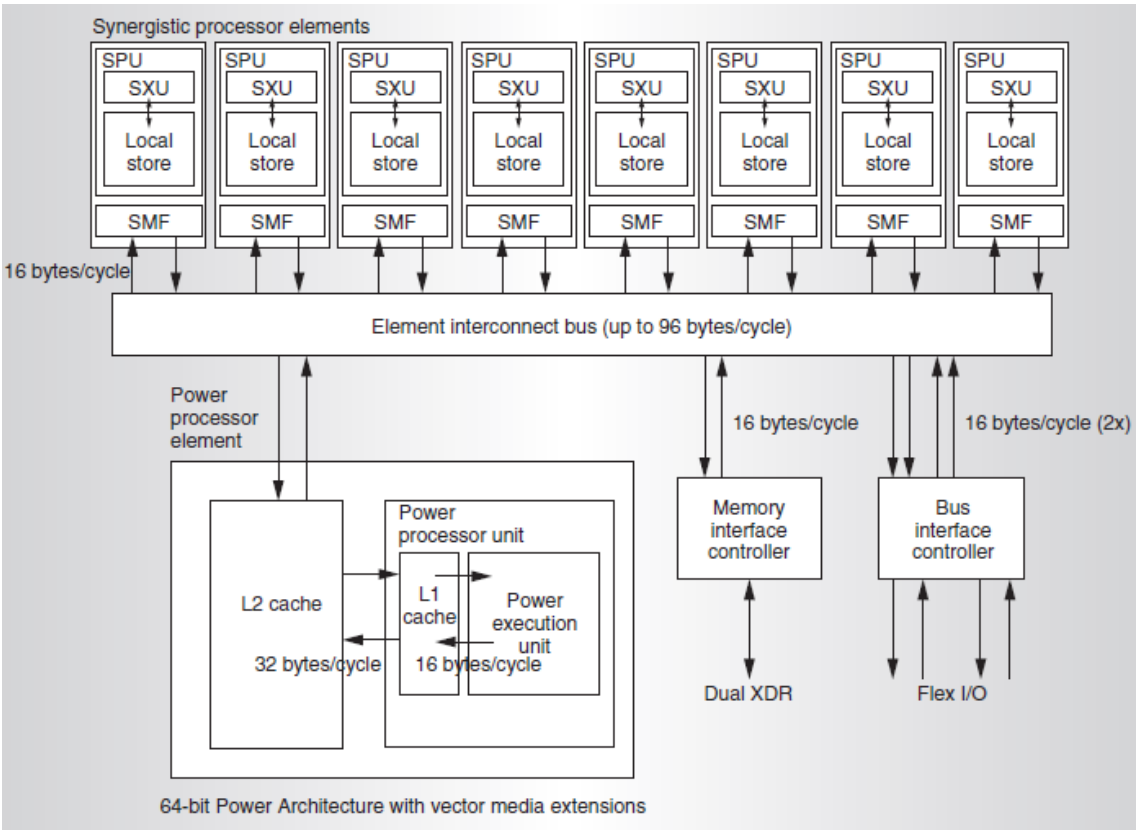

Figura 2.1: Arquitetura Cell (GSCHWIND, 2006).

Os PPEs e SPEs possuem acesso a um espaço comum de endereçamento, que inclui a memória principal, e uma extensão de endereços que correspondem à memória local de cada SPE, registradores de controle e dispositivos de entrada e saída. Além disso, os elementos de processamento utilizam os mesmos formatos de tipo de dados e semântica de operações, possibilitando um compartilhamento eficiente de dados entre eles. A comunicação entre os elementos de processamento e estruturas externas de I/O é feita através de um barramento circular de alto desempenho, chamado EIB (*Element Interconnect Bus*), que permite um acesso à memória totalmente coerente com a memória cache.

Apesar de, inicialmente, o processador ter sido concebido para ser utilizado em consoles de jogos (como o Playstation 3, da Sony), já foi utilizado em outros dispositivos eletrônicos de multimídia, tais como receptores digitais de televisões de alta definição. Pode-se dizer que a sua arquitetura possibilita avanços consideráveis no desempenho quando comparado com outros processadores, tornando-se assim esta arquitetura também vantajosa para uso em diversas aplicações científicas como processamento de dados sísmicos, dinâmica molecular e simulação de fluxos complexos (GREHS, 2009).

Além do CBE, existe hoje mais um processador em conformidade com a CBEA – o IBM PowerXCell 8i, que possui uma aritmética aprimorada para precisão dupla (64 bits) e utiliza memória DDR2 em vez de Rambus (XDR). Pode-se dizer que ambas as soluções fazem parte da CBEA (IBM, 2008-a).

Nas próximas seções são apresentadas com maiores detalhes as diferentes unidades que compõem a CBEA.

2.1 PowerPC Processor Element (PPE)

O PPE é um processador RISC (*Reduced Instruction Set Computer*) de 64 bits de propósito geral, com suporte para até duas *threads*. Sua arquitetura está em conformidade com a versão 2.02 da arquitetura PowerPC, criada em 1991 por uma aliança entre Apple, IBM e Motorola (bastante utilizada em sistemas embarcados de alto desempenho). Portanto, o PPE é capaz de executar programas escritos para outros processadores PowerPC.

O PPE é responsável pelo controle geral do sistema. É nessa unidade que executa o sistema operacional, que gerencia todas as aplicações a serem executadas no próprio PPE e nos SPEs. O PPE é constituído de duas unidades principais:

- PPU (*Power Processor Unit*), que executa propriamente as instruções de programa;
- PPSS (*Power Processor Storage Subsystem*), que trata as requisições de memória da PPU, bem como as requisições externas das SPEs e dispositivos de I/O.

A figura 2.2 ilustra de forma simplificada um diagrama de blocos do PPE.

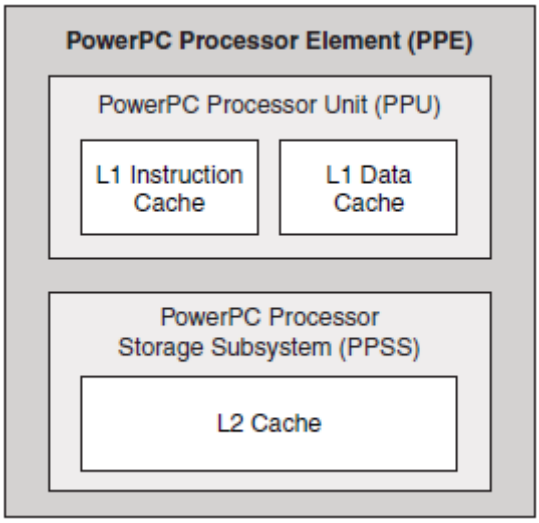

Figura 2.2: Diagrama de blocos do PowerPC Processor Element. (IBM, 2008-a)

2.1.1 PowerPC Processor Unit

A PPU é responsável por executar o conjunto de instruções da arquitetura PowerPC, assim como um conjunto de instruções SIMD (extensões multimídia). Ela possui dois conjuntos de bancos de registradores (um para cada *thread*) e mais um conjunto composto pelas seguintes unidades funcionais:

- IU (*Instruction Unit*) – Realiza tarefas como busca, decodificação e despacho de instruções. Contém a cache de instruções de nível 1 (L1), que é associativa em duas vias, com capacidade de 32KB, recarga automática em caso de erro e proteção por paridade. Cada linha da cache tem tamanho igual a 128 bytes.
- LSU (*Load and Store Unit*) – Realiza todos os acessos a dados, incluindo a execução de instruções de leitura e escrita. Contém a cache de dados de nível 1 (L1), que é associativa em quatro vias, do tipo *write-through*, com capacidade de 32KB e proteção por paridade. Cada linha dessa cache tem tamanho igual a 128 bytes.
- VSU (*Vector/Scalar Unit*) – Possui uma unidade de ponto flutuante (ou *Floating-Point* Unit – FPU) e uma unidade de extensão multimídia SIMD que opera sobre vetores de até 128 bits, denominada VXU (*Vector/SIMD Multimedia Extension Unit*).
- FXU (*Fixed-Point Unit*) – Executa operações de ponto fixo (inteiras), incluindo adição, multiplicação, comparação e instruções lógicas.
- MMU (*Memory Management Unit*) – É a unidade responsável por gerenciar a tradução de endereço para todos os acessos à memória.

A VXU é capaz de executar até oito operações de precisão simples a cada ciclo de relógio, alcançando um limite teórico de 25.6 GFLOPS a 3.2GHz. Operações de ponto flutuante com precisão dupla somente podem ser realizadas utilizando instruções escalares e a uma taxa máxima de duas operações por ciclo, alcançando assim o limite teórico de 6.4 GFLOPS (GREHS, 2009).

2.1.2 PowerPC Processor Storage Subsystem

O PPSS gerencia todos os acessos da PPU à memória, assim como as operações de coerência de memória (*snooping*) do EIB. Ele possui uma cache de nível 2 (L2) unificada (instruções e dados) que é associativa em oito vias, do tipo *write-back*, com capacidade de 512KB, código de correção de erros (ECC – *Error Correcting Code*) e linha de 128 bytes.

A comunicação entre a PPU e o PPSS se dá através de uma porta de escrita de 32 bytes (para requisições da MMU, da cache L1 de instruções e da cache L1 de dados) e uma porta de armazenamento de 16 bytes (para requisições da MMU e da cache L1 de dados). A interface entre o PPSS e o EIB possui barramentos de 16 bytes, tanto de leitura como de escrita.

2.2 Synergistic Processor Element (SPE)

Os SPEs são processadores independentes, cada um executando uma *thread* independente. Cada SPE inclui uma memória local (*Local Store* – LS) privada para acesso eficiente a dados e instruções. Além disso, cada SPE também possui acesso completo à memória compartilhada coerente, incluindo a área mapeada para I/O (GSCHWIND, 2006).

Cada SPE é um processador RISC de 128 bits especializado para aplicações escalares e SIMD, quando for exigida a computação intensiva de grandes volumes de dados. De forma geral, um SPE é basicamente constituído, como mostra a figura 2.3, de dois blocos: a SPU (*Synergistic Processor Unit*) e o MFC (*Memory Flow Controller*).

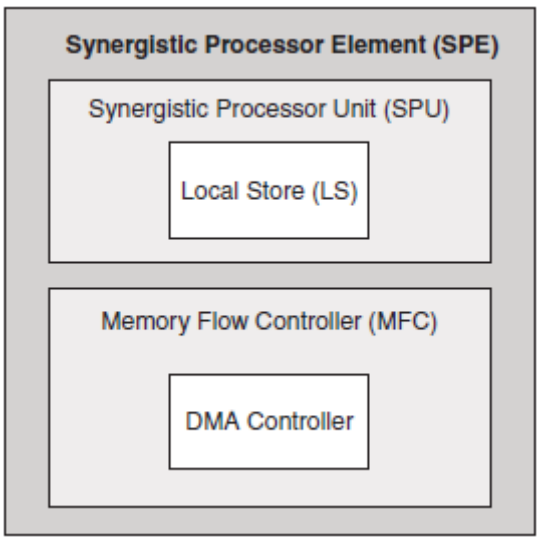

Figura 2.3: Diagrama de blocos do Synergistic Processor Element (IBM, 2008-a).

2.2.1 Synergistic Processor Unit

A SPU executa um conjunto próprio de instruções SIMD, denominado de *Synergistic Processor Unit Instruction Set Architecture* (SPU ISA), basicamente voltado para fazer uso de instruções SIMD sobre vetores. Mesmo assim, alguns operadores escalares também são suportados. Por se tratar de um conjunto inteiramente novo de instruções, o código gerado para uma SPU não é compatível com nenhum outro processador, nem mesmo com a PPU.

O projeto da SPU procurou reduzir a área por núcleo, aumentando o número de núcleos disponíveis e permitindo a operação em altas frequências pelo uso de *pipelines* de pequena profundidade. A plataforma não inclui mecanismos como predição de desvios, renomeação de registradores ou execução de instruções fora de ordem. A solução encontrada foi reduzir a complexidade da arquitetura onde possível, ficando sujeita às latências de decisões referentes a recursos básicos, como a grande memória local e o extenso banco de registradores (GSCHWIND, 2006).

A SPU possui uma memória local (LS), um banco de registradores (*SPU Register File Unit* - SRF) e uma unidade de execução chamada *Synergistic Execution Unit* (SXU).

A LS de uma SPU é uma memória de 256KB, protegida por códigos de correção de erros, que armazena todas as instruções e dados utilizados pela SPU. Possui apenas uma porta, suportando um acesso por ciclo de relógio. Os acessos podem ser originados de uma busca antecipada de instrução (*prefetch*), instruções de leitura/escrita ou acesso DMA (*Direct Memory Access*). A SPU realiza busca antecipada de instruções a 128 bytes por ciclo e acesso a dados a 16 bytes por ciclo, sendo que os dados precisam estar alinhados como palavra quádrupla (*quadword*). O acesso DMA é feito a uma taxa de 128 bytes por ciclo. Além disso, as transferências que possuem tamanho menor que o de uma palavra quádrupla são feitas suportando um ciclo do tipo *read-modify-write* na LS.

O SRF possui 128 entradas de 128 bits cada, sendo capaz de armazenar todos os tipos de dados suportados, além de endereços de retorno, resultados de comparações, entre outros. Além dos 128 registradores de propósito geral, há um registrador de 128 bits chamado FPSCR (*Floating-Point Status and Control Register*), que armazena informações sobre a última operação em ponto flutuante e exceções associadas.

A SXU é uma unidade de execução em ordem com dois *pipelines*, chamados respectivamente de *pipeline* par (*Even Pipeline*) e *pipeline* ímpar (*Odd Pipeline*). Ela é capaz de completar duas instruções por ciclo, uma em cada *pipeline*. As unidades internas da SXU são:

- SFS (*SPU Odd Fixed-Point Unit*) – Executa operações de deslocamento com granularidade de byte, rotação mascarada e *shuffle* em palavras quádruplas.
- SFX (*SPU Even Fixed-Point Unit*) – Executa instruções aritméticas e lógicas, deslocamentos e rotações do tipo SIMD, comparações em ponto flutuante e estimativas de inverso e raiz quadrada inversa.
- SFP (*SPU Floating-Point Unit*) – Executa instruções de ponto flutuante com precisão simples e dupla, conversões e multiplicações de inteiros e operações de byte. O hardware apenas suporta multiplicações inteiras de 16 bits. Multiplicações de inteiros de 32 bits são implementadas em software utilizando a multiplicação de 16 bits.
- SLS (*SPU Load and Store Unit*) – Executa instruções de leitura/escrita e HBR (*hint for branch*), além de controlar as requisições DMA à LS.
- SCN (*SPU Control Unit*) – Busca instruções e as envia para um dos dois pipelines, de acordo com o tipo de instrução. Também executa instruções de salto, arbitra o acesso à LS e ao SRF e realiza outras funções de controle.
- SSC (*SPU Channel and DMA Unit*) – Possibilita comunicação, tranferência de dados e controle para e a partir da SPU.

Uma visão geral da SPU, incluindo as unidades internas da SXU e suas alocações em relação aos *pipelines*, é mostrada na figura 2.4.

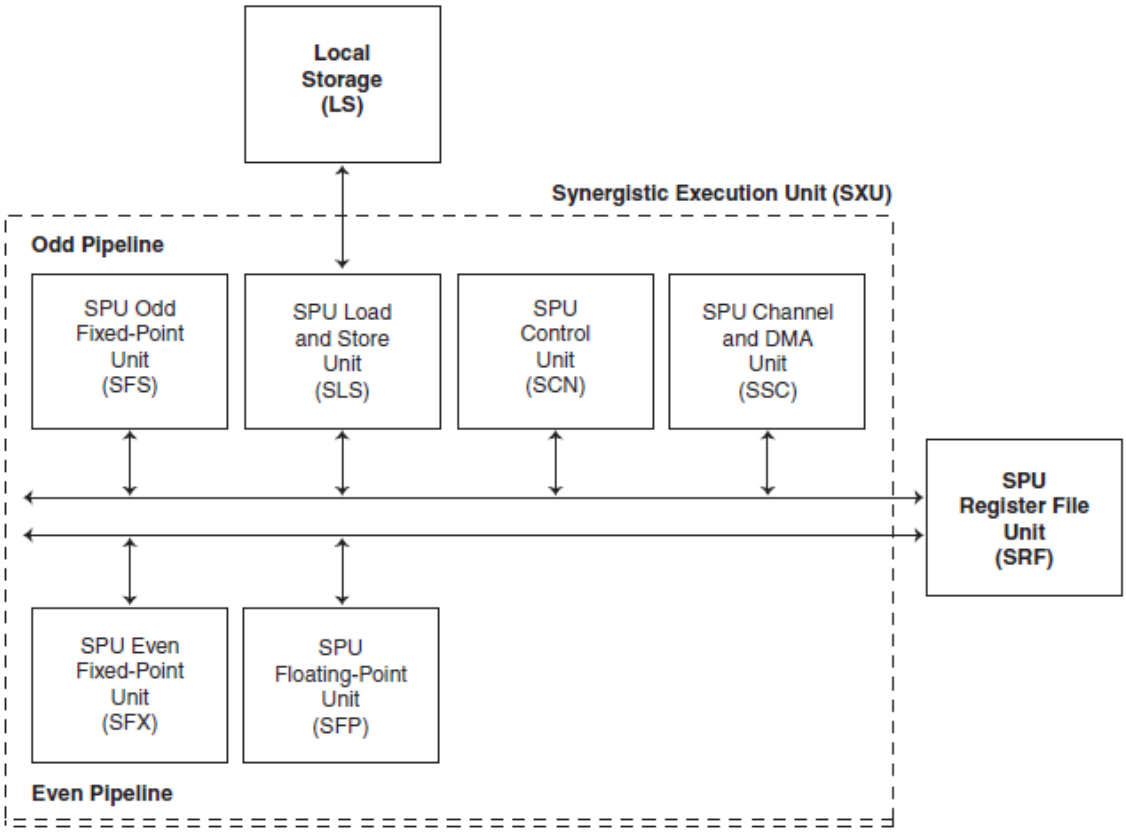

Figura 2.4: Principais unidades funcionais da SPU (IBM, 2008-a).

A SPU opera buscando blocos de instruções de 128 bytes da LS. Cada bloco é subdividido em 16 blocos de oito bytes cada um com uma ou duas instruções, chamados de *fetch groups*. A primeira instrução em cada *fetch group* tem origem em um endereço par, e a segunda em um endereço ímpar.

Uma instrução no *fetch group* está pronta para execução quando não há conflito de recursos e as dependências de registradores são satisfeitas. Neste caso, essa instrução é despachada para um *pipeline* de execução. Dependendo do tipo de instrução, ela vai para o *pipeline* par ou ímpar. Pode ocorrer um despacho duplo quando ambas as instruções no *fetch group* atual estão prontas, com a primeira pertencendo ao *pipeline* par e a segunda pertencendo ao *pipeline* ímpar.

A falta de uma unidade de predição de desvios, memória cache e hardware para execução especulativa tem um forte impacto no desempenho de códigos de controle intensivo, portanto deve-se evitar executar esse tipo de código nos SPEs. A organização da SXU também atribui as tarefas de otimização ao compilador e ao próprio programador. Entretanto, as regras de escalonamento dos *pipelines* são relativamente simples, e é possível gerar escalonamentos estáticos de alta qualidade em software de maneira direta. Além disso, isso cria um ambiente determinístico, o que favorece aplicações de tempo-real (GREHS, 2009).

2.2.2 Memory Flow Controller

A SPU não acessa diretamente a memória principal, dispositivos de sistema ou outros elementos de processamento. Para tanto, o MFC serve como interface entre a SPU e esses elementos, utilizando-se do EIB. Seu papel principal é o de uma interface entre o domínio da LS e o domínio da memória principal, por meio de um controlador DMA que move instruções e dados entre essas duas memórias. O MFC ainda suporta sincronização entre as memórias e funções de comunicação com o PPE, outros SPEs e dispositivos. A figura 2.5 mostra um diagrama de blocos do MFC.

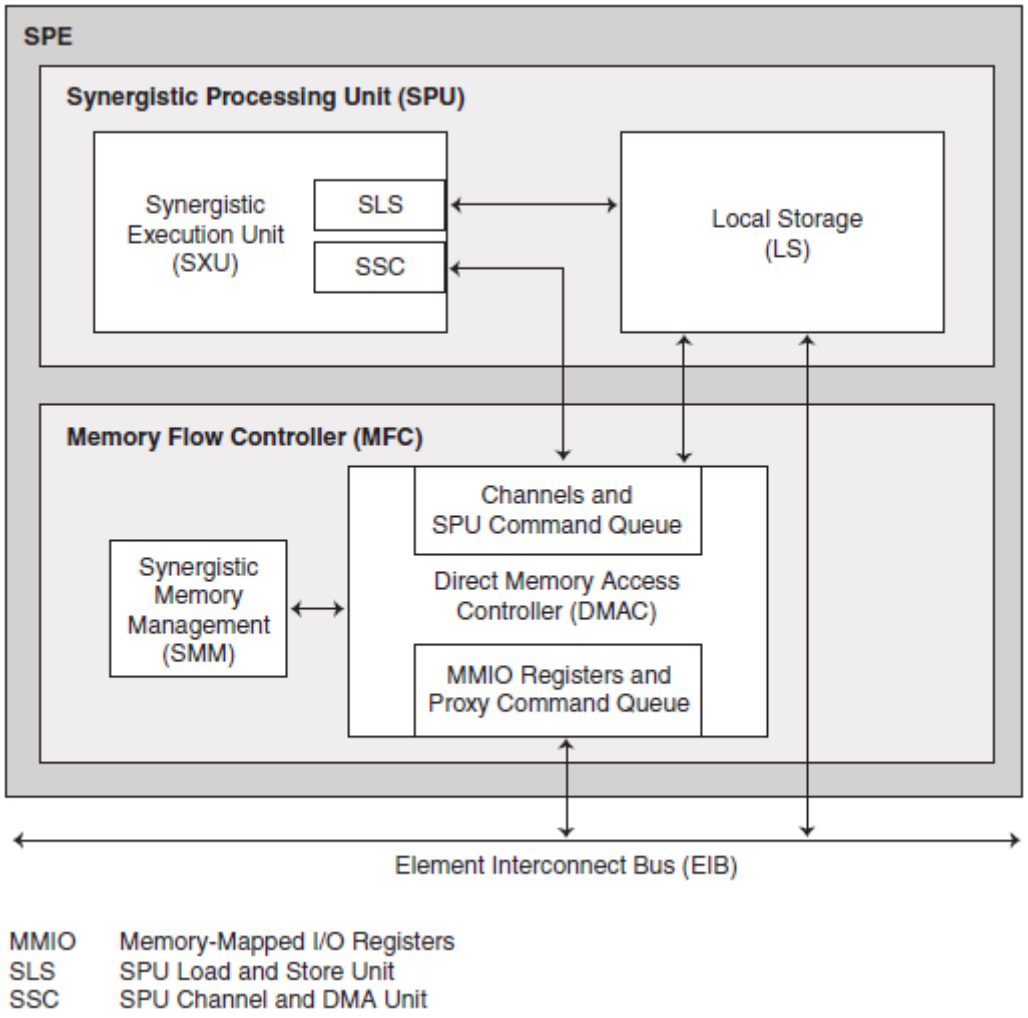

Figura 2.5: Diagrama de blocos do MFC (IBM, 2008-a).

2.2.2.1 Canais e comandos MFC

O software que roda no SPE se comunica com a memória principal, o PPE, outros SPEs e dispositivos através de canais. Na CBEA, canais são interfaces unidirecionais de passagem de mensagens que suportam comandos e mensagens de 32 bits. Cada SPE acessa seus próprios canais através de instruções especiais para utilizar comandos MFC.

O software que roda no PPE, outros SPEs e dispositivos podem acessar funções das interfaces de canais do SPE através dos registradores MMIO (*memory-mapped I/O*) no espaço da memória principal.

Comandos MFC servem para iniciar transferências DMA, verificar o status do controlador DMA, realizar comunicação entre processos, entre outras funções. Os comandos MFC que implementam transferências DMA entre a LS e a memória principal são chamados de comandos DMA.

2.2.2.2 Controlador DMA

O controlador DMA do MFC possui uma fila de comandos, que armazena os comandos enviados através das interfaces de canais, e uma fila de *proxy*, que armazena comandos enviados pelos registradores MMIO. A execução desses comandos não ocorre necessariamente em ordem, mas se necessário podem ser usados comandos que forçam a execução ordenada.

O MFC executa comandos DMA de forma autônoma, permitindo que a SPU continue executando paralelamente enquanto ocorrem transferências DMA envolvendo a LS. Cada controlador DMA pode iniciar até 16 transferências DMA independentes com a sua correspondente LS.

Cada transferência DMA pode mover até 16KB com um comando. É possível a realização de transferências de tamanho igual a 1, 2, 4, 8 ou então múltiplos de 16 bytes.

Também são suportadas transferências DMA em lista. Uma lista DMA é uma sequência de elementos que, juntamente com um comando de DMA *list*, especificam uma sequência de transferências DMA entre uma área da LS e áreas possivelmente descontínuas da memória principal.

2.2.2.3 Mailboxes e Signalling

Mailboxes são mecanismos do MFC que permitem comunicação entre processos através de mensagens curtas. Cada interface de canal em cada SPE suporta dois *mailboxes* independentes: um para mensagens de um SPE para o PPE (*outbound mailboxes*) e uma para mensagens no caminho inverso (*inbound mailboxes*). SPEs podem trocar mensagens desse tipo entre si, de forma a permitir o acesso aos registradores MMIO um do outro.

Um *inbound mailbox* é capaz de armazenar até quatro mensagens em um buffer do tipo FIFO. Uma leitura em um *mailbox* vazio bloqueia a SPU até que uma mensagem seja recebida. Se uma mensagem for recebida enquanto o *mailbox* estiver cheio, a última mensagem é sobrescrita.

Um *outbound mailbox* somente suporta uma mensagem ao PPE por vez, sendo a SPU bloqueada em uma tentativa de escrever em um *mailbox* cheio. O PPE precisa requisitar a mensagem a um determinado SPE para que ela seja lida. É importante verificar a disponibilidade de mensagens no SPE antes de requisitar uma mensagem, pois não há bloqueio da PPU se o *mailbox* estiver vazio. Uma mensagem inválida é recebida nesse caso. Para evitar *polling* e, consequentemente, tráfego desnecessário no EIB, um dos dois *outbound mailboxes* do MFC é de um tipo especial, chamado *interrupt outbound mailbox*. Esta permite que uma tarefa na PPU seja bloqueada quando uma mensagem é escrita pela SPU.

O PPE também pode enviar mensagens ao SPE por meio de sinalização (*signal-notification channels*), um mecanismo parecido com os *mailboxes*. O MFC possui dois registradores de 32 bits para esse propósito. Esses registradores não podem ser escritos pela SPU local como um *inbound mailbox*, porém são limpos sempre que são lidos.

2.3 Element Interconnection Bus (EIB)

O EIB é o caminho de comunicação para comandos e dados entre todos os elementos de processamento e os controladores de memória e I/O. O EIB suporta operações SMP (*Symmetric Multi-Processing*) simétricas e totalmente coerentes com a memória.

O EIB consiste de quatro anéis de dados de largura igual a 16 bytes. Cada anel transfere 128 bytes (uma linha de cache PPE) por vez. Múltiplas transferências podem ocorrer concorrentemente em cada anel, incluindo mais de 100 requisições DMA entre a memória principal e os SPEs. A banda interna máxima do EIB é de 96 bytes por ciclo de relógio.

2.4 Programando em Cell

A ferramenta de desenvolvimento para Cell (*IBM Cell Broadband Engine SDK*), atualmente na versão 3.1, permite programar em linguagem C, tanto para o PPE quanto para os SPEs. O SDK (*System Development Kit*) é compatível com os sistemas operacionais RHEL (*Red Hat Enterprise Linux*) 5.2 e Fedora Core 9.

Uma vez que o PPE e os SPEs possuem conjuntos de instruções diferentes, é necessário utilizar compiladores distintos (*ppu-gcc* e *spu-gcc*, respectivamente) e gerar arquivos executáveis separados para o PPE e para cada programa diferente dos SPEs. O PPE é responsável por carregar os binários correspondentes para cada SPE, assim como comandar o início de sua execução.

Como não possuem gerenciamentos internos complexos, como execução fora de ordem, predição de desvios e cache, mesmo com a ajuda do compilador, o programador ainda precisa realizar algum controle manual, como alinhamento de dados e transferências DMA. Além disso, como a linguagem C não tem suporte para o conjunto de instruções do SPE, para utilizar instruções específicas é necessário utilizar uma série de comandos adicionais chamados de *SPU intrinsics*.

2.5 Considerações finais sobre a tecnologia CBEA

A arquitetura CBEA proporciona um alto grau de paralelismo, utilizando multiprocessamento e operações SIMD. Isso favorece aplicações que atuam sobre grandes volumes de dados, incluindo aplicações de multimídia como codificação de áudio e de vídeo.

3 ESTIMATIVA DE MOVIMENTO EM VÍDEOS

Até 20 anos atrás, a maioria dos equipamentos de vídeo era projetada primariamente para operar com vídeo analógico. A utilização de vídeo digital ocorria somente em aplicações profissionais, como edição de vídeo. Com o desenvolvimento da tecnologia de microeletrônica e a evolução dos processadores, nas últimas décadas, vídeos em formato digital estão cada vez mais presentes no nosso cotidiano, seja na televisão de alta definição, no computador ou mesmo no telefone celular (JACK, 2007).

Apesar desses avanços tecnológicos, a codificação de vídeo digital é ainda um processo bastante custoso computacionalmente. Pode-se dizer que com o aumento da demanda por vídeos de alta definição em tempo-real, cada vez mais técnicas otimizadas que acelerem esse processo são necessárias. (HÜSEMANN, 2010).

A seguir alguns dos principais conceitos utilizados por codificadores de vídeo são apresentados de forma mais detalhada.

3.1 Espaço de cores YCbCr

Para que seja possível representar um vídeo em cores, é necessário um mecanismo capaz de representar informações de cor. Uma imagem monocromática somente precisa de um valor para indicar o nível de brilho de cada amostra espacial. Imagens coloridas, por outro lado, precisam de pelo menos três valores por pixel para representar cores com precisão. O método escolhido para representar brilho (luminância ou luma) e cor é descrito como um espaço de cores.

Existem vários espaços de cores padronizados para uso em aplicações diferentes. Por exemplo, o espaço RGB, muito utilizado em sistemas computacionais, representa cores através de três números que indicam as proporções relativas das três cores primárias aditivas da luz: vermelho, verde e azul. Em sistemas de vídeo, no entanto, o espaço de cores mais popular é o YCbCr, ou YUV (RICHARDSON, 2003).

Sua adoção baseia-se no fato de que o sistema visual humano é mais sensível a variações de luminância do que a variações de cor. No espaço RGB, as três cores têm a mesma importância e são normalmente armazenadas com a mesma resolução. É possível, entretanto, representar uma imagem em cores de forma mais eficiente separando a luminância da informação de cor, utilizando uma resolução melhor para a luminância.

O espaço de cores YCbCr utiliza-se dessa estratégia para reduzir o espaço de representação. Neste modelo de representação, a componente Y é a luminância, sendo calculada a partir de uma média ponderada dos valores R, G e B.

A informação de cor é representada por componentes de crominância (ou simplesmente croma) Cr, Cb e Cg, onde cada componente equivale à diferença entre R, G ou B e Y. Como a soma de todos os componentes de croma (Cr + Cg + Cb) é uma constante, apenas duas das três componentes precisam ser armazenadas ou transmitidas, e a terceira é calculada a partir das outras duas.

A característica do YCbCr de permitir resoluções diferentes para componentes luma e croma deu origem a diferentes formatos de amostragem. Na prática, os três formatos mais comuns são o 4:4:4, o 4:2:2 e o 4:2:0, retratados na figura 3.1. No formato 4:4:4, as três componentes possuem a mesma resolução e, portanto, existe uma amostra de cada componente em cada posição de pixel. No formato 4:2:2, as componentes croma possuem a mesma resolução vertical do que a luminância, porém metade da resolução horizontal. No formato 4:2:0, as componentes Cr e Cb possuem metade da resolução horizontal e metade da resolução vertical da componente Y. O olho humano tem pouca percepção desta perda de informação, o que permite, ao se utilizar o formato 4:2:0, uma economia significativa do espaço necessário para representação dos pixels (RICHARDSON, 2003).

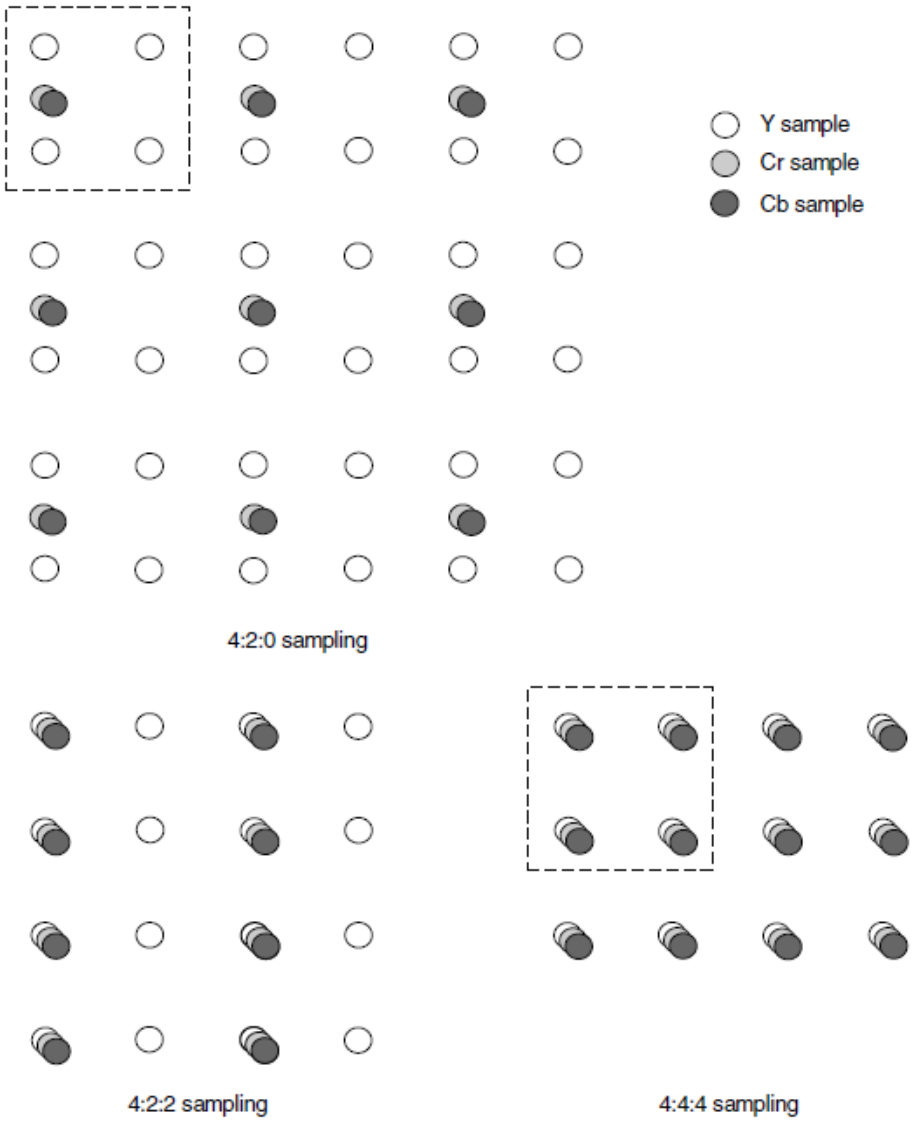

Figura 3.1: Formatos de amostragem YCbCr (RICHARDSON, 2003).

3.2 Estimativa de Movimento

Vídeos digitais tipicamente demandam uma grande taxa de bits para transmissão. Como exemplo, basta considerar uma sequência com definição de televisão (SD – *Standard Definition*), mesmo utilizando um formato 4:2:2 e subamostragem temporal (luminância a 13.5 MHz e crominância a 6.75 Hz para 30 fps), precisa de aproximadamente 216 Mbits por segundo. Para possibilitar armazenamento e transmissão de vídeo digital, é necessário utilizar sofisticadas técnicas de compressão (RICHARDSON, 2003).

A compressão de dados se baseia em encontrar e remover redundâncias do vídeo, reduzindo assim o número de informações necessárias para poder representá-lo. Alguns tipos de dados podem ser comprimidos de forma eficaz sem nenhum tipo de perda. Infelizmente, padrões de compressão de imagens sem perdas não alcançam taxas de compressão vantajosas para vídeos. Por isso, para atingir taxas maiores, os codificadores de vídeo fazem uso de algoritmos de compressão com perdas, o que significa que o dado recuperado na decodificação não é exatamente igual ao original. Como o olho humano não é muito sensível, estas perdas acabam não sendo percebidas.

De forma simplificada, pode-se dizer que a maioria dos métodos de compressão de vídeo explora dois tipos de redundância: redundância espacial, que existe entre pixels próximos em uma imagem, e redundância temporal, que ocorre entre quadros temporalmente adjacentes na sequência. Para reduzir a redundância espacial, são utilizados métodos de compressão de imagens estáticas. Para a redundância temporal, empregam-se algoritmos de estimativa de movimento sobre um modelo temporal.

O modelo temporal permite a análise de similaridade entre quadros temporalmente distintos a fim de se aproveitar de informações já transmitidas. O quadro que pode ser montado a partir de informações previamente transmitidas é chamado de quadro predito (ou quadro estimado). Subtraindo-se o quadro predito do quadro atual, produz-se o que se chama de quadro residual. O quadro residual é codificado e enviado para o decodificador, de forma a corrigir os erros produzidos pela reconstrução do vídeo utilizando-se apenas o quadro predito. Quanto melhor for a predição, menos bits serão necessários para o quadro residual, pois menos erros serão detectados (JACK, 2007).

O quadro predito pode ser formado a partir de um ou mais quadros passados ou futuros. O processo de predição pode ser melhorado significativamente compensando o movimento entre os quadros de referência e o quadro atual. Para que isso seja possível, é necessário estimar o movimento de cada **bloco** de um dado tamanho (M x N pixels), considerando as relações de posicionamento de blocos similares em um quadro de referência, gerando os chamados **vetores de movimento**. Um vetor de movimento representa a informação de diferença relativa entre a posição atual e a posição onde um bloco muito similar foi localizado em um quadro já transmitido. Quanto maior o tamanho do bloco, menos vetores de movimento precisam ser enviados ao decodificador, porém maior será a diferença residual correspondente. Muitos padrões importantes de codificação (incluindo MPEG-1, MPEG-2, MPEG-4 Visual, H.263 e H.264) utilizam como unidade básica para a predição de movimento o **macrobloco**, que corresponde a uma região de 16 x 16 pixels. A figura 3.2 mostra um quadro residual gerado sem compensação de movimento (esquerda), um com compensação em blocos 16 x 16 (meio) e um que utiliza blocos 4 x 4 (direita). Tons de cinza mais escuros significam uma diferença negativa, e tons mais claros uma diferença positiva.

Figura 3.2: Compensação de movimento (RICHARDSON, 2003).

Para encontrar os vetores de movimento, é necessário fazer para cada bloco da imagem atual uma busca na imagem de referência pelo bloco correspondente, o que exige um grande número de comparações. Essas comparações consomem muito tempo, e por isso vários algoritmos de busca têm sido propostos, nos últimos anos, visando a diminuir o número de comparações necessárias sem que isso implique em um aumento significativo na taxa de bits (HÜSEMANN, 2008).

Uma técnica bastante utilizada é a de restringir a área de busca a uma região de um determinado tamanho, chamada **janela de busca**, que está centrada na posição atual do bloco, pois se assume que, na maioria das vezes, o movimento não é rápido o suficiente para exceder os limites daquela região (figura 3.3). Outra estratégia para agilizar o processo é calcular o movimento apenas para a componente de luma, considerando que os vetores correspondentes para as componentes de crominância são os mesmos.

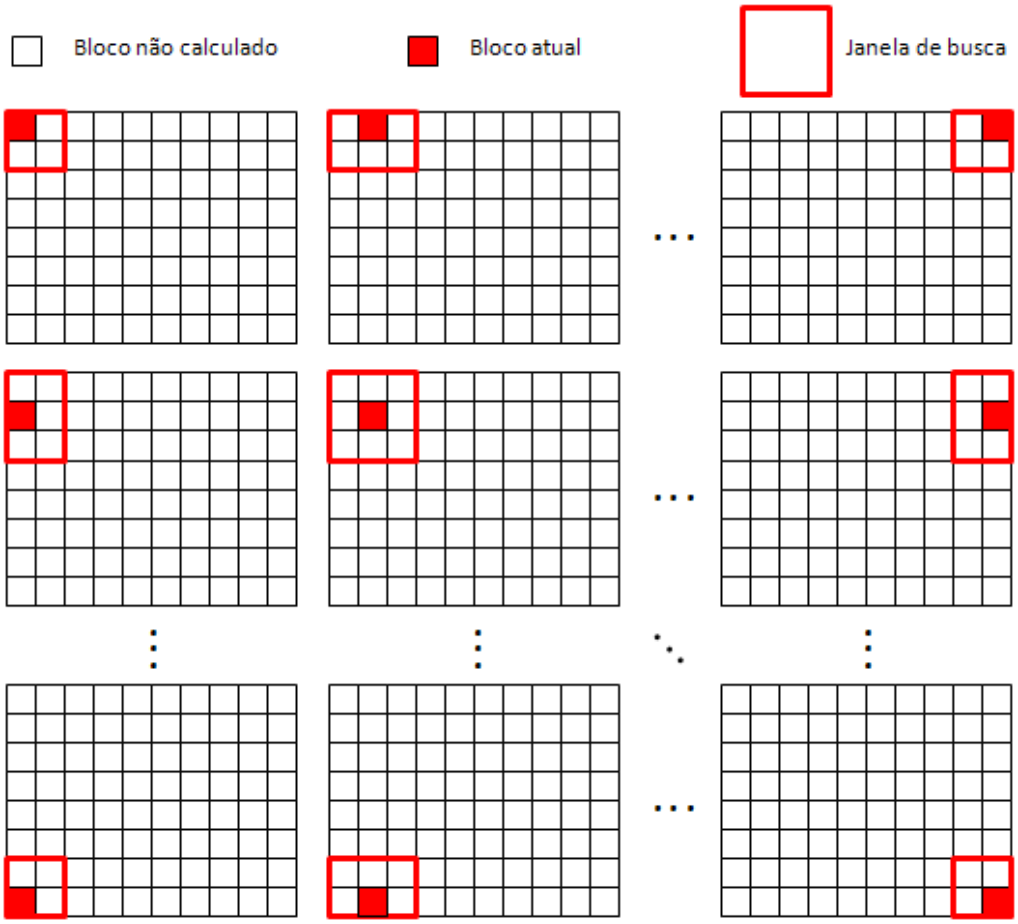

Figura 3.3: Janela de busca centrada nos bloco atual.

Na maioria das vezes, o bloco encontrado não é exatamente igual ao que se procura. Os algoritmos devem assim tentar encontrar um bloco na imagem de referência que possua a maior similaridade possível com o bloco atual. Para isso, é necessária uma medida de similaridade que seja eficaz e computacionalmente eficiente, pois seu cálculo será realizado para comparações feitas entre blocos em uma grande quantidade de posições possíveis. Dentre os mecanismos de cálculo de similaridade mais utilizados estão o SAD (*Sum of Absolute Differences*), que é a soma das diferenças absolutas dos valores de cada pixel, o MSE (*Mean Square Error*), que é o erro quadrático médio, o MAD (*Mean Absolute Difference*), que é a diferença absoluta média, entre outros.

Um esquema geral do processo de cálculo de vetor de movimento para um bloco pode ser visto na figura 3.4. Na figura, um bloco similar ao do quadro atual está sendo procurado ao redor de uma janela de busca que se estende de [-p,p], centrada no próprio bloco.

Quando o bloco de maior similaridade é encontrado, a diferença de posição relativa (deslocamento) do bloco encontrado em relação ao original é considerado como o vetor de movimento detectado.

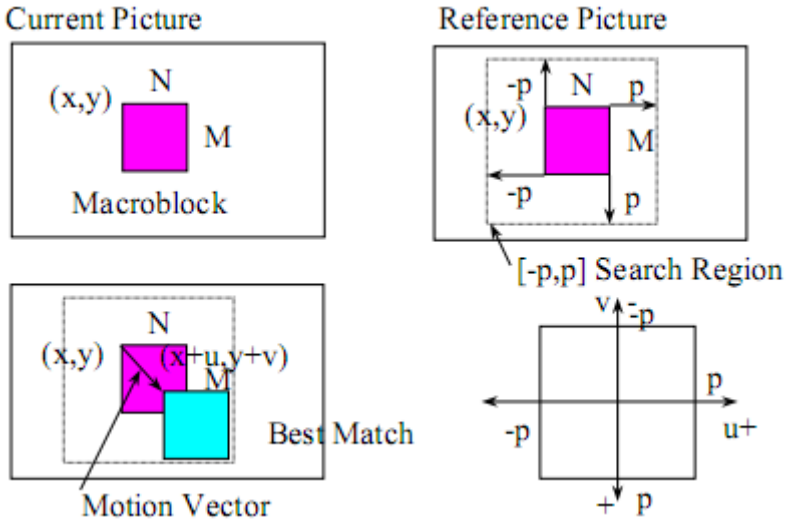

Figura 3.4: Mecanismo de cálculo do vetor de movimento (GECKLE, 2000).

Como normalmente se tem uma banda limitada, em geral o critério utilizado para comparar algoritmos na área de codificação de vídeos é a qualidade final da imagem utilizando-se uma taxa de bits fixa, e não a taxa de bits em si. Como qualidade de imagem é um conceito subjetivo e difícil de analisar computacionalmente, os algoritmos de codificação de vídeo, em geral, consideram critérios determinísticos, como o PSNR (*Peak Signal-to-Noise Ratio*), dado em decibéis (dB) (RICHARDSON, 2003).

3.3 Algoritmo de busca

A maioria dos algoritmos de busca começa a procurar pelo bloco em alguma posição inicial e, a partir daí, realizam a procura pelas proximidades de acordo com regras e padrões de busca próprios até encontrar um erro mínimo. A posição inicial pode ser a própria posição do bloco na imagem atual ou pode ser determinada por um ou mais vetores preditores, que podem ser escolhidos de diversas formas.

Em (HÜSEMANN, 2010) é proposto um algoritmo de estimativa de movimento que tem como objetivo acelerar o processo sem que se tenha uma perda significativa na qualidade da imagem final.

O algoritmo escolhido utiliza um padrão de busca simples, conhecido como padrão de diamante pequeno. Para cada estágio do padrão de diamante pequeno, cinco candidatos são analisados em uma topologia em forma de cruz (centro, acima, abaixo, esquerda e direita) com resolução de um pixel, como mostra a figura 3.5.

Após a realização do cálculo de similaridade para todos os cinco possíveis blocos, o candidato com a maior similaridade (menor SAD) é escolhido, tornando-se o centro do diamante para uma nova busca. A busca termina quando o a maior similaridade for a do bloco na posição central.

É importante notar que, neste algoritmo, apenas no primeiro estágio é preciso calcular o SAD das cinco posições de bloco. Para os estágios seguintes, apenas três são necessários (indicados como quadrados pretos na figura), uma vez que dois deles já foram calculados no estágio anterior. Isso contribui para a redução do número de iterações necessárias para se localizar o bloco de maior similaridade, aumentando o desempenho do algoritmo de forma global.

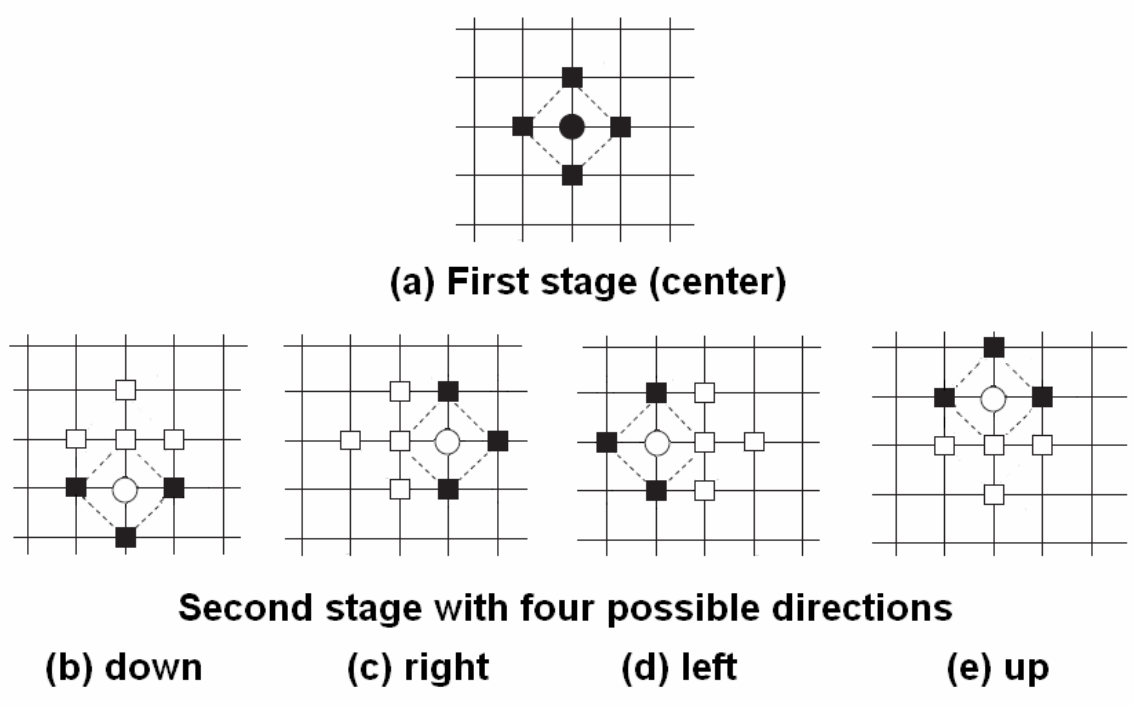

Figura 3.5: Padrão de diamante pequeno (HÜSEMANN, 2010).

Outra consideração feita na definição deste algoritmo foi de que macroblocos espacialmente próximos geralmente possuem uma grande correlação. Sendo assim, valores já calculados de posições vizinhas podem ser usados como preditores para a posição inicial de cada procedimento de busca, reduzindo o número de iterações necessárias para encontrar o bloco de similaridade. Além disso, o uso desses preditores ajuda a reduzir o erro causado por SADs mínimos locais. Dependendo da configuração, podem ser usados diferentes preditores.

Na prática, como a procura por vetores de movimento é feita linha a linha, da esquerda para a direita e de cima para baixo para cada macrobloco da imagem atual, três

valores que podem ser considerados como “bons candidatos”. Estes representam os vetores encontrados para o bloco à esquerda, para o bloco acima e para o bloco acima e à direita.

No algoritmo de (HÜSEMANN, 2010), a posição inicial de busca é calculada pela média dos vetores já calculados dentre esses três (desde que já tenham sido determinados). Na prática, a definição real do número de preditores a serem considerados vai depender da respectiva localização do bloco alvo dentro da janela de busca, uma vez que blocos localizados nas bordas superior, à direita e à esquerda não têm todos os seus vizinhos. Uma ilustração disso é apresentada na figura 3.6.

Outros algoritmos, como o PMVFAST (TOURAPIS, 2001), também utilizam esses preditores.

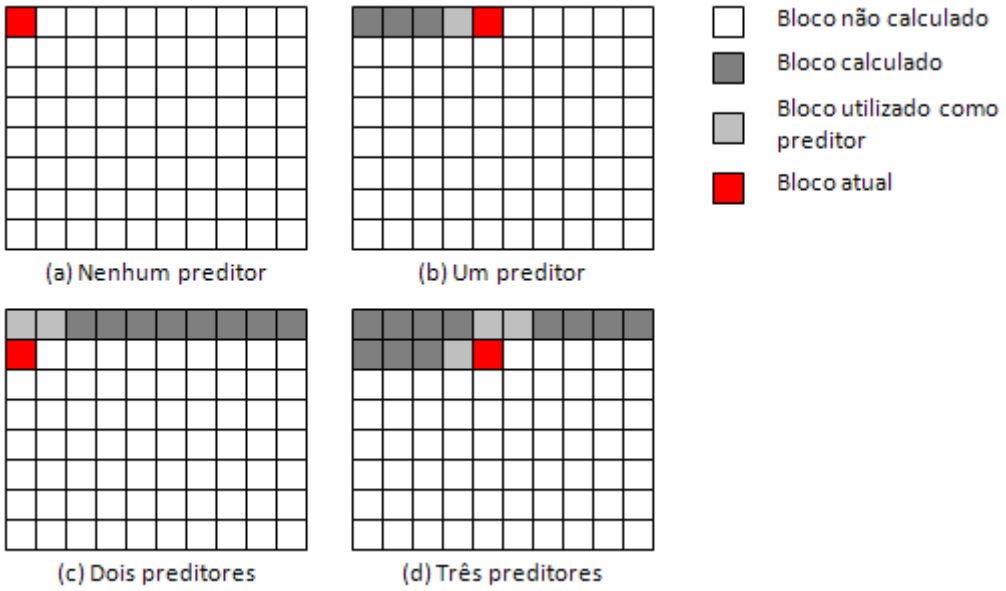

Figura 3.6: Preditores de movimento.

3.4 Cálculo de similaridade

Como foi dito anteriormente, para determinar o bloco de maior similaridade é necessário um método de cálculo de similaridade. Por ser especialmente simples, utilizando apenas somas e subtrações, o método de SAD é o mais utilizado.

O SAD resultante da comparação de um bloco A de tamanho NxN localizado na posição (x,y) dentro do quadro atual com um bloco B localizado a um deslocamento (v_x,v_y) em relação a A no quadro de referência é definido como (TOURAPIS, 2001):

$$SAD(v_x, v_y) = \sum_{m,n=0}^{N-1} \left| I_t(x+m, y+n) - I_{t-i}(x+v_x+m, y+v_y+n) \right| \qquad (1)$$

Além disso, vale destacar que este método, por sua simetria, pode ser facilmente implementado utilizando operações SIMD.

No trabalho de (HÜSEMANN, 2010), para facilitar a utilização de operações vetoriais, propõe-se que todas as estruturas utilizadas no algoritmo utilizem uma

granularidade de 64 bits, o que permite a leitura de um grupo de 8 bytes (8 x 8 = 64 bits), sendo obtidas assim com um único acesso à memória.

A busca é realizada com blocos de tamanho 16 x 16, e o padrão de diamante utilizado tem resolução de um pixel. Outra estratégia utilizada para redução do número de operações é a de utilizar o cálculo da similaridade utilizando uma versão subamostrada do bloco atual (no caso com tamanho 8 x 8). Isso, além de diminuir o número de subtrações em quatro vezes, ainda possibilita que cada linha ou coluna do bloco seja representada exatamente por um valor de 64 bits.

A fim de se evitar a realização do procedimento de subamostragem e remontagem dos vetores de 64 bits a cada novo bloco pesquisado na imagem de referência, essa imagem é inicialmente dividida em quatro imagens subamostradas. Na prática, esse processo de subamostragem classifica os pixels de acordo com suas linhas e colunas, dependendo se forem pares ou ímpares. A figura 3.6 exemplifica o processo de subamostragem com uma imagem hipotética de 4 x 4 pixels.

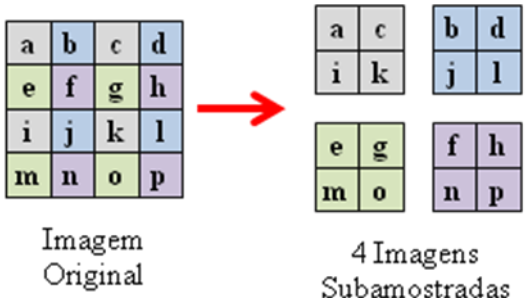

Figura 3.7: Subdivisão da imagem de referência.

O bloco do quadro de referência é implementado com quatro vetores de 64 bits, chamados de vetores lineares, cada um contendo os pixels do bloco que estiverem em uma das subamostragens da imagem de referência. Dois desses vetores são alinhados com a direção horizontal, sendo os outros dois alinhados com a direção vertical, como mostrado na figura 3.7. O alinhamento dos dados nesses vetores favorece o uso de instruções SIMD.

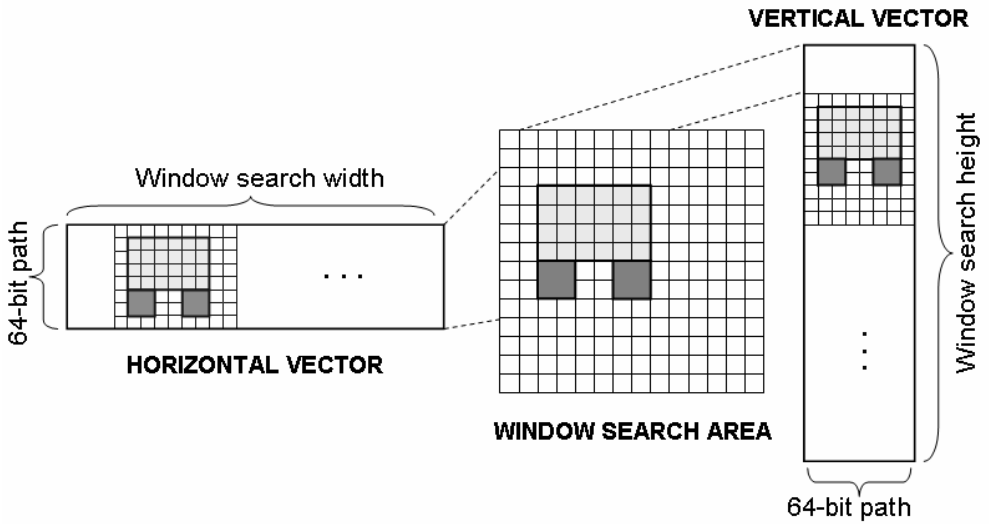

Figura 3.8: Relação entre vetores lineares e janela de busca (HÜSEMANN, 2010).

Para possibilitar o cálculo dos cinco SADs correspondentes ao padrão de diamante, devido às paridades dos pixels, são necessários três vetores diferentes (um para a posição central, um para os deslocamentos verticais e um para os deslocamentos horizontais). Entretanto, com exceção da primeira iteração do diamante, a posição central nunca precisa ser calculada. Isso permite que, para cada nova posição do diamante, apenas dois desses vetores (alinhados com direções diferentes) sejam necessários.

Assim sendo, o deslocamento do diamante exige, para um dos vetores, apenas a atualização de uma posição de memória e de um ponteiro. O vetor, entretanto, que estiver alinhado com uma direção distinta do deslocamento determinado (horizontal ou vertical) precisa ser inteiramente atualizado. Mesmo assim o ganho relacionado com o uso de instruções SIMD alinhadas com estes vetores ainda justifica esta metodologia.

A janela de busca é implementada apenas com valores que indicam os seus limites máximos e mínimos.

3.5 Considerações finais sobre estimativa de movimento

Neste capítulo foi apresentado um algoritmo de estimativa de movimento em sequências de vídeo. Esse algoritmo é importante, pois ajuda a aproveitar redundâncias temporais nas imagens, reduzindo significativamente a taxa de bits de saída do codificador. Esse processo exige um grande número de comparações entre blocos, possuindo um elevado custo computacional.

O próximo capítulo discute a paralelização e implementação do algoritmo citado no processador CBE.

4 DESENVOLVIMENTO DA APLICAÇÃO

4.1 Paralelização

No processo de estimativa de movimento, para cada bloco, realiza-se uma busca ao redor de certa região do quadro de referência. Esse procedimento de busca e comparação que se realiza é praticamente independente para cada novo bloco. A única interdependência de dados que pode se perceber ocorre quando do uso de preditores de posição inicial, pois estes precisam ser calculados previamente.

Considerando essa independência, pode-se levantar uma idéia inicial de solução paralela para o algoritmo de estimativa de movimento, baseada na realização paralela dos cálculos de vetor para cada bloco. Nesta solução, o cálculo de cada bloco é atribuído a um SPE, repetindo-se o processo até que todos os vetores estejam calculados.

Essa abordagem, no entanto, não aproveita a localidade dos dados na imagem, exigindo um grande número de transferências de memória e causando perda de desempenho. Além disso, no caso da arquitetura escolhida, os preditores são parte fundamental do algoritmo, e sua remoção em favor do paralelismo causaria uma perda de desempenho ainda maior (TOURAPIS, 2001).

Como alternativa, buscou-se uma resposta intermediária, que combinasse os ganhos de paralelismo e da utilização de preditores. A solução encontrada foi dividir a imagem em partes iguais, ou **subquadros**, reduzindo o problema a encontrar blocos similares dentro de imagens menores. Dessa forma, é possível que cada unidade de execução paralela processe todos os vetores associados a um desses subquadros.

Para simplificar o algoritmo, foi descartada a interdependência temporal entre subquadros. Ou seja, no algoritmo proposto são considerados apenas preditores que estejam dentro do mesmo subquadro. A figura 4.1 exemplifica isso considerando uma imagem dividida em quatro subquadros.

Basicamente essa limitação tende a causar efeitos apenas nos blocos localizados nas bordas de cada subquadro. O impacto prático dessa solução será analisado posteriormente.

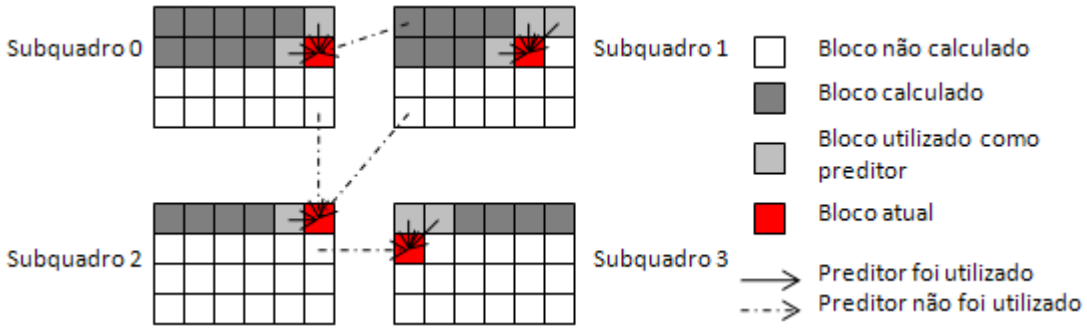

Figura 4.1: Subquadros e preditores.

Conforme já citado, a janela de busca é centrada no bloco atual, o que faz com que ela fique incompleta nas regiões de fronteiras entre subquadros. Para contornar esse problema, é necessário que os subquadros de referência contenham extensões de dimensão (bordas) correspondentes à porção da janela de busca que pode exceder os seus limites.

Esse princípio de extensão dos subquadros pode ser o observado na figura 4.2. Na figura, considera-se uma janela de busca de tamanho 48x48 e um quadro de tamanho 256x128, sendo este dividido em oito subquadros. Em cada subquadro, é mostrada a janela de busca correspondente a um de seus blocos.

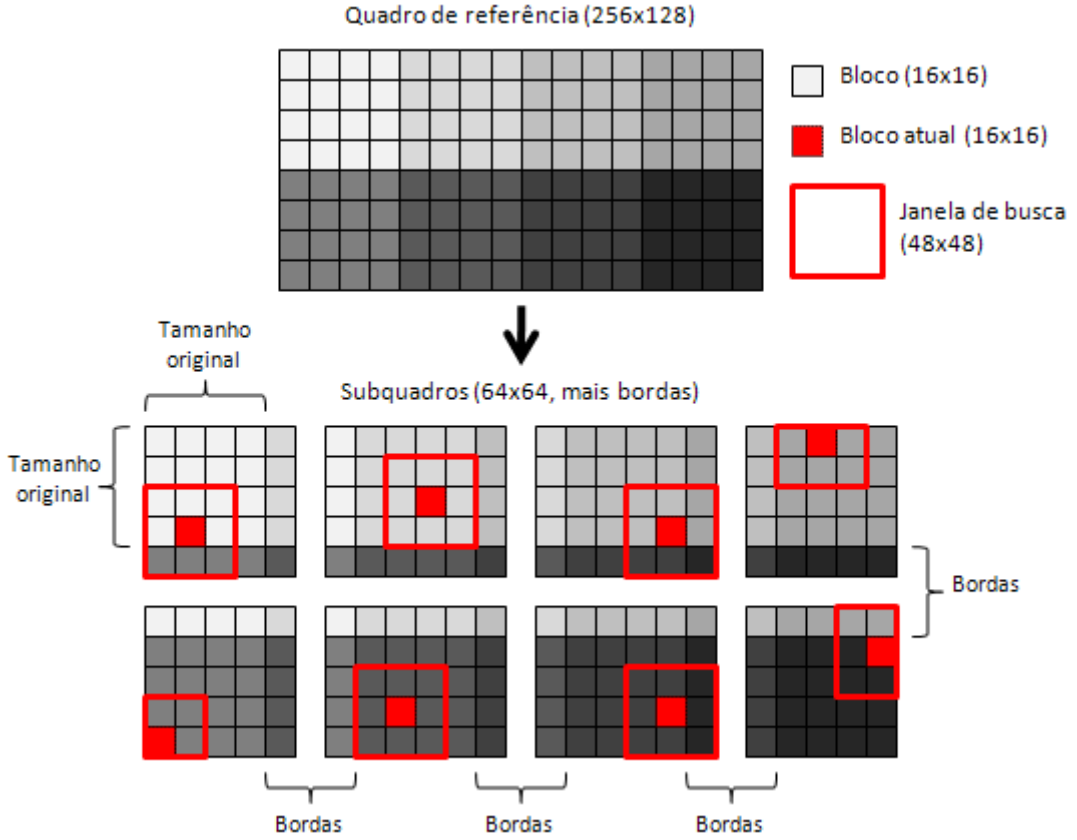

Figura 4.2: Subquadros de referência.

A memória necessária para o armazenamento de cada subquadro pode ser calculada de acordo com:

$$M_{cur} = \frac{H \times W}{D_y \times D_x} \times \frac{1}{F_s} \times A \quad (2)$$

$$M_{ref} = \left(\frac{H}{D_y} + 2B_y\right) \times \left(\frac{W}{D_x} + 2B_x\right) \times \frac{N_s}{F_s} \times A \quad (3)$$

onde:

- M_{cur} = memória necessária para o subquadro atual;
- M_{ref} = memória necessária para o subquadro de referência;
- W = largura do vídeo;
- H = altura do vídeo;
- D_x = número de divisões de subquadro para a largura;
- D_y = número de divisões de subquadro para a altura;
- B_x = tamanho da extensão (borda) em largura;
- B_y = tamanho da extensão (borda) em altura;
- F_s = fator de subamostragem;
- N_s = número de subamostragens do subquadro;
- A = número de alinhamentos de dados.

No caso do algoritmo proposto, o fator de subamostragem F_s é sempre igual a 4, o número de alinhamentos A é sempre 2 (um horizontal e um vertical) e o número de subamostragens N_s é igual a 4.

É importante que se faça esse cálculo, pois os subquadros constituem a maior parte da memória necessária para cada SPE. Essa memória, como já foi dito, é limitada a 256KB compartilhados entre código e dados.

Tanto o algoritmo original quanto a versão paralela precisam passar, para cada novo quadro, por uma etapa e montagem de estruturas (quadros ou subquadros subamostrados) antes de começar a estimativa de movimento. Como essas são funções genéricas, essa etapa foi atribuída ao PPE.

Assim que essa etapa inicial de montagem das imagens subamostradas e dos subquadros é terminada, a estimativa de movimento em si é iniciada.

Seguindo o modelo proposto, os cálculos referentes a cada subquadro foram delegados aos SPEs, sendo realizados paralelamente.

4.2 Comunicação entre PPE e SPEs

A comunicação entre os processadores é feita através de mensagens, utilizando *mailboxes*. Ao todo, são utilizadas três mensagens diferentes:

- SPE_FREE – Enviada por um SPE ao PPE, através de seu *outbound mailbox*, para avisar que está livre para receber uma nova tarefa;
- SPE_GO – Enviada pelo PPE para um SPE, através do *inbound mailbox* correspondente, para que se inicie a estimativa de movimento em um subquadro;

- SPE_KILL – Enviada pelo PPE para um SPE, através do *inbound mailbox* correspondente, para que sua execução seja terminada.

Todas as outras transferências de dados entre PPE e SPEs são feitas por requisições DMA, feitas pelos SPEs. Na prática, os SPEs recebem por DMA os parâmetros de vídeo, os endereços dos subquadros e, finalmente, os próprios subquadros. Além, disso, os vetores de movimento calculados são enviados por DMA para o PPE.

4.3 Algoritmo do PPE

Na primeira vez que o módulo de estimativa de movimento é chamado, o PPE inicialmente aloca memória para todas as estruturas necessárias para subquadros e vetores de movimento e inicializa os SPEs. Essa inicialização inclui a carga do programa, que é o mesmo para todos os SPEs, e o comando de início de execução (cada um em uma *thread*).

A figura 4.3 mostra um fluxograma em passos gerais do algoritmo executado pelo PPE para cada novo quadro a ser calculado.

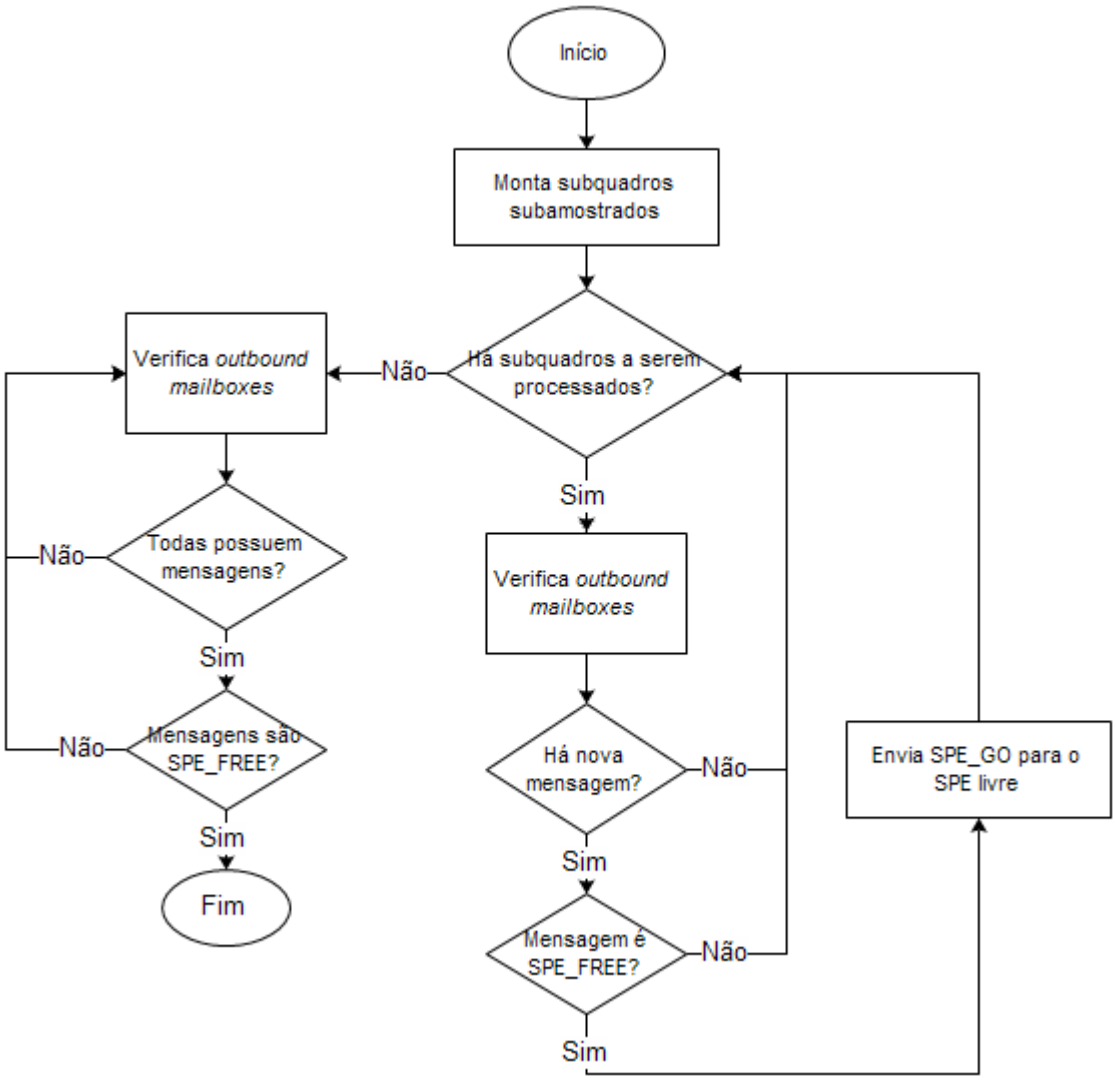

Figura 4.3: Fluxograma do PPE.

Depois de calculados, os vetores ficam armazenados no PPE, podendo ser utilizados pelo codificador até que seja requisitado o cálculo de movimento para um novo quadro.

4.4 Algoritmo dos SPEs

Um SPE inicia a execução de seu código com o pedido para o PPE de uma estrutura que descreve alguns parâmetros do vídeo, como altura e largura. A partir desses parâmetros, é possível realizar todas as alocações de memória necessárias. Após a alocação de memória, inicia-se o laço principal do algoritmo do SPE. A figura 4.4 mostra um fluxograma em passos gerais do algoritmo executado por cada um dos SPEs.

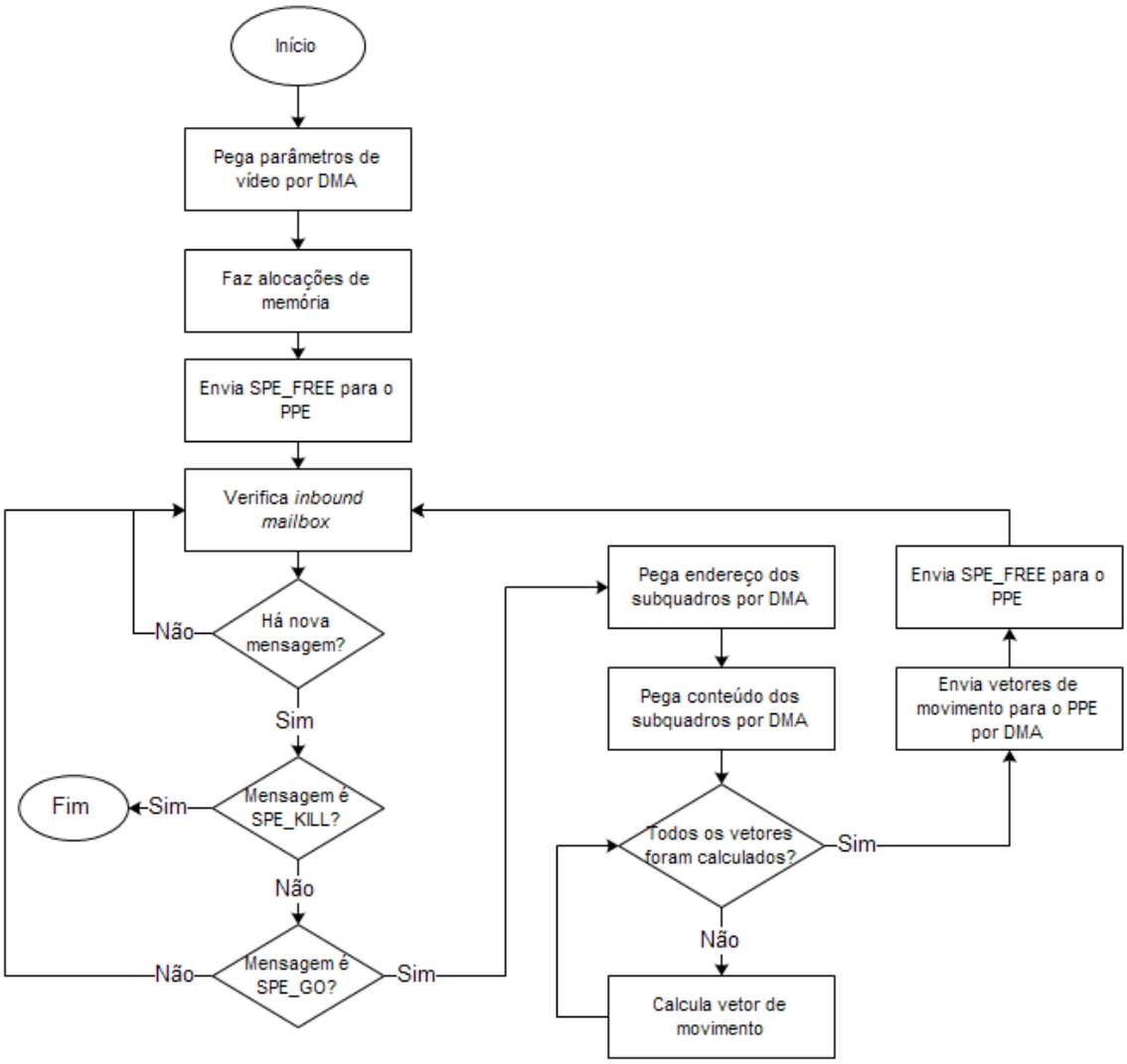

Figura 4.4: Fluxograma dos SPEs.

4.5 Uso de SIMD

Conforme apresentado na equação (1), o SAD entre dois blocos é calculado pela soma das diferenças absolutas dos valores dos pixels desses blocos.

Analisando-se o conjunto de instruções do SPE, encontrou-se uma instrução, denominada *spu_absd*, que realiza, de uma única vez, o cálculo das diferenças absolutas entre os bytes de dois vetores de 128 bits (16 bytes em paralelo).

Os cálculos de SAD no algoritmo proposto são realizados sobre macroblocos subamostrados, de tamanho 8 x 8, que são armazenados em vetores alinhados em 64 bits (8 bytes). Considerando-se a arquitetura do SPE, um bloco inteiro pode ser considerado como quatro vetores de 16 bytes (16 x 8 bits = 128 bits).

A partir dessa manipulação dos dados, pode-se então utilizar a instrução *spu_absd* para calcular as diferenças absolutas entre os pixels de dois blocos com apenas quatro instruções.

Entretanto, na prática, a realização da soma dessas diferenças absolutas não ocorre de maneira tão direta, pois a instrução *spu_absd* gera uma saída vetorial, e não escalar (figura 4.5).

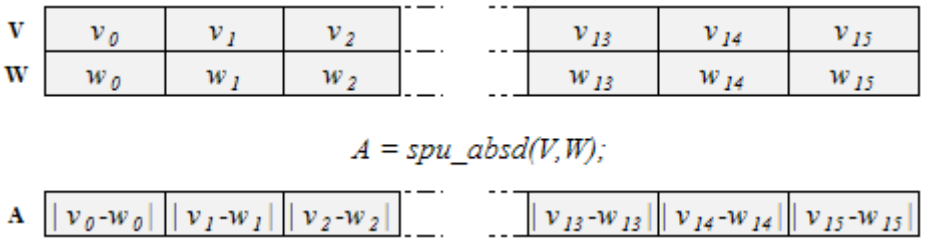

Figura 4.5: Instrução *spu_absd*.

Para o cálculo total do SAD, de fato, são utilizadas as seguintes instruções SIMD do SPU ISA:

- **r = spu_absd(a,b)** – Calcula, a diferença absoluta entre os bytes correspondentes dos vetores ***a*** e ***b***, armazenando o resultado no vetor ***r***;
- **r = spu_rlqwbyte(a,n)** – Rotaciona a palavra de 128 bits ***a*** de ***n*** bytes para a esquerda, armazenando o resultado no vetor ***r***;
- **r = spu_add(a,b)** – Soma os elementos dos vetores ***a*** e ***b***, armazenando o resultado no vetor ***r***;
- **r = spu_sumb(a,b)** – Realiza somas de 4 em 4 bytes com os vetores ***a*** e ***b***, armazenando o resultado em valores de 16 bits no vetor ***r***;
- **x = spu_extract(a,n)** – Armazena na variável escalar ***x*** o valor do elemento ***n*** do vetor ***a***.

Para facilitar o entendimento do algoritmo, é necessário que se explique, em especial, o efeito da instrução *spu_sumb*. É importante perceber que, apesar de essa instrução receber como entrada dois vetores de 16 bytes, a sua saída é um vetor composto por oito valores de 16 bits. A execução dessa instrução totaliza em oito valores, cada um sendo o resultado da soma de 4 bytes dentro um dos vetores. A figura 4.6 mostra como são realizadas essas somas para dois vetores ***a*** e ***b***, e como os resultados são armazenados no vetor ***r***. Na figura, os elementos mais claros em ***r*** correspondem às somas provenientes do vetor ***a***, enquanto as mais escuras correspondem às do vetor ***b***.

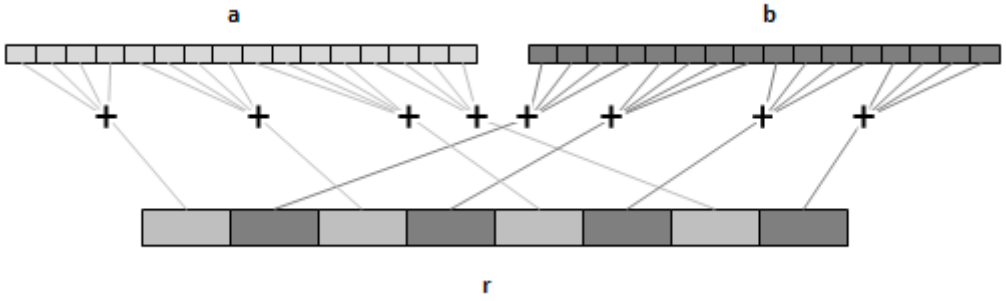

Figura 4.6: Instrução *spu_sumb*.

O cálculo de SAD entre dois blocos, considerando que cada bloco é composto de quatro vetores SIMD de 128 bits, é realizado separadamente dois a dois por vez, somando-se os resultados obtidos em cada operação de soma para se obter o SAD total.

Cada soma individual de vetores é calculada, inicialmente, através de uma mesma operação *spu_absd*, retornando em um vetor as diferenças absolutas relativas (16 valores de diferenças com resolução de byte). Em seguida, sçao tomados dois desses vetores resultantes, que são somados com a instrução *spu_sumb* (figura 4.6), resultando em um vetor com oito elementos de 16 bits.

Finalmente, para somar os elementos desse vetor, utiliza-se uma técnica que consiste em realizar somas sucessivas do vetor com cópias dele mesmo, porém rotacionadas. A cada iteração, o passo de rotação é aumentado, até que se obtenha a soma total de todos os elementos. A figura 4.7 mostra o funcionamento dessa técnica. Vale lembrar que o vetor a ser somado é composto de valores de 16 bits, porém a instrução utilizada para as rotações, *spu_rlqwbyte*, recebe como parâmetro um número de bytes (8 bits) a serem rotacionados.

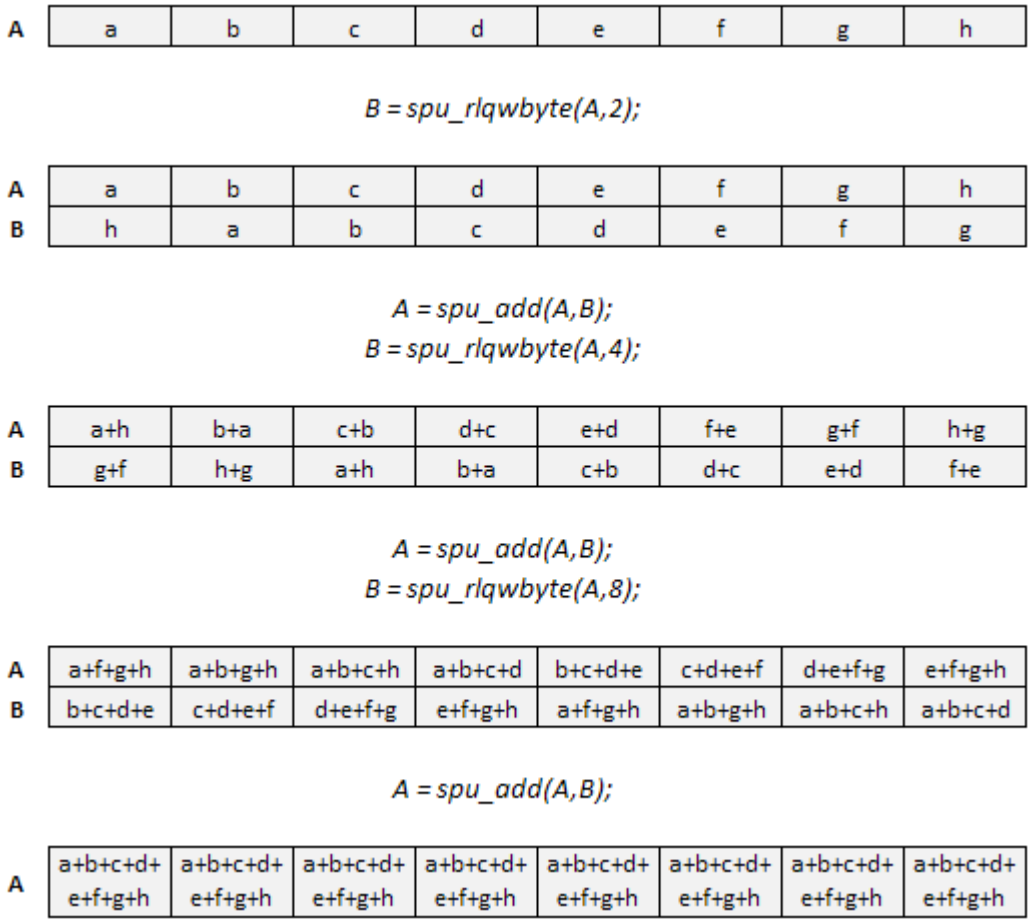

Figura 4.7: Soma dos elementos do vetor.

A instrução *spu_extract*, por fim, é utilizada para extrair uma das somas totais no vetor resultante.

Como se pode perceber, o uso de instruções SIMD no SPE exige que os dados estejam alinhados com palavras de 128 bits. Assim sendo, os vetores do algoritmo proposto, que operava inicialmente com dados alinhados em 64 bits, teve de ser adaptado para agrupar dois vetores em uma estrutura nova de 128 bits antes do cálculo do SAD ser realizado.

4.6 Considerações finais sobre o desenvolvimento

Neste capítulo, foi discutida uma forma de se paralelizar o algoritmo de estimativa de movimento descrito no capítulo 3. A paralelização foi realizada baseada na divisão dos quadros em imagens menores, chamadas de subquadros, aproveitando a independência dos cálculos de vetores de movimento em blocos espacialmente distintos. Finalmente, o algoritmo de cálculo de similaridade entre blocos (SAD) foi aprimorado a partir do uso de instruções SIMD.

5 RESULTADOS

5.1 Ambiente

Durante a primeira etapa do projeto, os testes do algoritmo paralelizado foram realizados em nível de simulação. Para tanto, foi utilizado um computador pessoal Core 2 Quad 2,8GHz, 4GB DDR2 FSB 800MHz, com sistema operacional Fedora Core 9, onde foi instalada a ferramenta de desenvolvimento Cell SDK versão 3.1. O simulador utilizado para validação do algoritmo foi o ambiente *IBM Full System Simulator for the Cell Broadband Engine*, que emula a arquitetura completa de um processador CBE.

Posteriormente, os algoritmos validados na primeira etapa foram portados para uma plataforma real que utilizava o CBE como microprocessador principal. A plataforma utilizada foi um console Sony PlayStation 3 (PS3), onde foi instalado o sistema Ubuntu 9.04. Como esse console estava fisicamente localizado em outra universidade, a transferência de aplicativos, operação e coleta de dados foram realizadas a partir de acesso via *ssh* (*Secure Shell*).

Todos os resultados dos ensaios descritos neste capítulo foram obtidos a partir desta plataforma (console PS3).

A plataforma PS3 não foi projetada para desenvolvimento de aplicativos externos e por isso apresenta algumas limitações de uso. Uma delas reside no fato de que nem todos os SPEs do CBE estão disponíveis para utilização. Isso, pois um deles é desativado durante a produção, enquanto que outro é reservado para tarefas do sistema operacional. Assim sendo, na prática, em um PS3 restam apenas 6 SPEs para programas de usuário.

5.2 Metodologia

Os desenvolvimentos realizados nestes experimentos práticos utilizaram como base o software de referência oficial do codificador H.264[1], denominado de JM (*JVT Model*), versão 16.2, que é fornecido pela entidade JVT (*Joint Video Team*) mantenedora de software das normas H.26x (SÜHRING, 2009). Com isso pode-se contar com uma mesma base confiável para experimentos de comparação de qualidade de imagem e de desempenho.

[1] O H.264 é o mais recente padrão de codificação de vídeo da família H.26x, definido pela organização ITU (*International Telecommunication Union*).

O software JM é bastante modular, o que facilita a adaptação e testes de algoritmos individualmente. Considerando-se a proposta em questão, apenas o módulo de estimativa de movimento do JM precisou ser adaptado.

Durante o desenvolvimento, quatro versões distintas de módulos de estimativa de movimento foram geradas, cada uma correspondendo a uma etapa diferente da implementação do algoritmo:

I. Básica: versão não paralela, onde o algoritmo executa inteiramente no PPE sobre a imagem inteira;

II. Multi-quadro: versão onde a imagem é dividida em 16 subquadros, todos estes manipulados no PPE;

III. Paralela padrão: versão paralelizada com imagem dividida em 16 subquadros, onde o PPE apenas prepara as estruturas de dados, e os SPEs processam os subquadros, sem o uso de instruções SIMD;

IV. Paralela otimizada: versão paralelizada com imagem dividida em 16 subquadros, onde o PPE apenas prepara as estruturas de dados, e os SPEs processam os subquadros, com o uso de instruções SIMD.

Para o levantamento de dados práticos comparativos, essas versões foram aplicadas sobre distintas sequências de vídeo. Foram utilizadas sequências de vídeo bem conhecidas da comunidade científica: *City*, *Crew*, *Harbour* e *Ice*, todas em tamanho 4CIF (704x576 pixels). A escolha desses vídeos foi feita visando avaliar na prática o desempenho do algoritmo para vídeos com características diferentes de movimentação. Com exceção da sequência *Crew*, onde foram codificados 43 quadros, todas as sequências foram codificadas até o centésimo quadro. Também para avaliar o desempenho do algoritmo em diferentes situações, foram definidas quatro taxas de bit de saída diferentes, que variam de 1500 a 3000 Kbps.

Em todos os testes feitos, o tamanho utilizado para a janela de busca foi de 48x48 pixels.

A resolução dos vídeos, a dimensão da janela de busca e o número de subquadros foram escolhidos considerando-se as limitações de capacidade de memória local (LS) de cada SPE, ou seja, o valor total não poderia ultrapassar o limite de 256KB. Para esses valores, a memória necessária para armazenar os dois subquadros (atual e referência), com base nas equações (2) e (3), é de aproximadamente 86 KB.

Os ensaios foram realizados em três etapas. A primeira etapa teve como objetivo a medição prática dos ganhos percentuais, obtidos comparando-se o algoritmo em sua versão básica com as outras versões aprimoradas (multi-quadro, paralela normal e paralela otimizada). Foram considerados nesta comparação os valores do tempo de execução específico da função de estimativa de movimento para cada uma das versões do algoritmo.

Na segunda etapa, os tempos de estimativa de movimento foram analisados de forma mais particularizada, de forma a segmentar o tempo gasto com preparação de estruturas e cálculo de movimento. A preparação de estruturas corresponde ao processo de subamostragem dos quadros e montagem dos subquadros, que é sempre executado no PPE. O cálculo de movimento corresponde ao algoritmo de estimativa de movimento em si (busca e comparação do bloco de pixels, buscando-se o mais similar), que roda em paralelo nos SPEs nas versões paralelas (normal e otimizada).

Na terceira etapa, foi medido o tempo de execução médio utilizado para calcular todos os vetores de movimento de cada subquadro. Esse tempo, no caso das versões paralelas, inclui o tempo gasto com transferências de estruturas entre SPE e PPE. A versão básica do algoritmo não utiliza subquadros e, por isso, não é considerada nesta etapa.

Para a realização das medidas temporais foi tomada como base a unidade de "ticks" de relógio. Para a plataforma de testes utilizada, esses "ticks" são contados a uma freqüência base de 79,8MHz (WYGANOWSKI, 2008).

Na prática, para obter o tempo de execução no PPE, utilizou-se a instrução *__mftb*, que retorna o valor atual do registrador de base de tempo da PPU (*Time Base Register*), que é incrementado internamente segundo a frequência base citada. Subtraindo-se o valor contido nesse registrador no início da execução de uma etapa do valor desse registrador no final da etapa, pode-se calcular o tempo total de execução de cada módulo do software.

Para os tempos de execução nos SPEs, o procedimento é análogo. Cada SPE possui um registrador (*SPU Decrementer*), que é decrementado à mesma frequência que o registrador de base de tempo do PPE. Foi utilizada a instrução *spu_write_decrementer* para ajustar um valor inicial a esse registrador, lendo o valor final com a instrução *spu_read_decrementer*. Como esse valor é decrementado, ao contrário do registrador de base de tempo, a subtração feita para se obter o tempo de execução precisa ser no sentido inverso.

Além das medidas de tempo de execução (desempenho), foi também realizada uma comparação dos resultados finais de qualidade sobre a imagem, em termos de PSNR, a fim de se analisar o impacto da omissão de preditores nas bordas dos subquadros. Medidas de PSNR sobre as imagens finais são calculadas e fornecidas pelo próprio software JM.

Nas seções seguintes, são apresentados e analisados os resultados obtidos em desempenho e qualidade.

5.3 Medidas de desempenho

5.3.1 Total da estimativa de movimento

As medidas de ganho de desempenho de cada uma das versões aprimoradas do algoritmo (multi-quadro, paralela normal e paralela otimizada) foram calculadas e resumidas na tabela 5.1.

O ganho foi calculado dividindo-se o tempo de execução da estimativa de movimento de cada versão aprimorada pelo tempo da estimativa na versão básica.

Tabela 5.1: Desempenho em função do tempo de estimativa de movimento.

Sequência	Taxa (Kbps)	Ganho em relação à versão básica (%)		
		Multi-quadro	Paralela normal	Paralela otimizada
CITY (704x576)	1500	69,96	73,80	86,70
	2000	27,31	37,74	45,65
	2500	25,44	35,34	46,16
	3000	23,31	34,00	43,99
CREW (704x576)	1500	20,64	24,81	38,79
	2000	18,03	20,83	32,80
	2500	19,59	22,10	35,37
	3000	19,67	22,71	36,15
HARBOUR (704x576)	1500	28,16	39,16	50,76
	2000	22,79	31,72	43,41
	2500	29,35	40,43	51,63
	3000	28,41	39,07	49,88
ICE (704x576)	1500	22,20	25,64	40,13
	2000	20,06	26,04	38,43
	2500	25,42	34,46	46,22
	3000	26,51	34,78	47,48
MÉDIA TOTAL		26,68	33,92	45,85

Os dados obtidos apontam para ganhos médios que variam de 26% a 45%, que correspondem aos ganhos reais obtidos com cada versão do algoritmo. Essa medição, no entanto, não reflete de forma individual o ganho obtido com o paralelismo nos SPEs. Isso, pois o tempo gasto com as preparações de estruturas, que são executadas no PPE para todas as versões do algoritmo, não pode ser desconsiderado. Além disso, por se tratarem de estruturas diferentes (visto que a versão básica trabalha com quadros, e as demais com subquadros), os tempos referentes à montagem de estruturas não é o mesmo em todas as versões.

5.3.2 Preparação de estruturas e cálculo de movimento

Na prática, a fim de se obter uma medida real do ganho obtido com o paralelismo deste algoritmo, é preciso segmentar as medidas de tepo de execução em dois tempos diferentes: tempo com preparação de estruturas e tempo de cálculo de movimento, como na tabela 5.2.

Conforme foi dito na seção 5.2, a preparação de estruturas corresponde ao processo de subamostragem dos quadros e montagem dos subquadros (PPE), e o cálculo de movimento corresponde à estimativa de movimento propriamente dita (SPEs).

Tabela 5.2: Tempos de preparação de estruturas e estimativa de movimento.

Sequência	Taxa (Kbps)	Tempo gasto com	Tempo (milhões de "ticks")			
			Básica	Multi-quadro	Paralela normal	Paralela otimizada
CITY (704x576)	1500	*preparação*	76,762	40,669	51,922	52,074
		cálculo	109,740	69,066	55,385	47,821
	2000	*preparação*	75,595	41,189	46,443	48,327
		cálculo	63,367	67,964	54,447	47,080
	2500	*preparação*	72,454	40,645	46,543	46,491
		cálculo	62,974	67,317	53,521	46,165
	3000	*preparação*	69,845	40,499	45,666	46,032
		cálculo	62,079	66,489	52,783	45,590
CREW (704x576)	1500	*preparação*	30,864	17,725	20,156	20,062
		cálculo	43,661	44,050	39,555	33,634
	2000	*preparação*	28,927	17,404	19,819	20,292
		cálculo	41,335	42,125	38,329	32,617
	2500	*preparação*	29,633	17,435	19,742	19,702
		cálculo	40,149	40,915	37,411	31,847
	3000	*preparação*	29,214	17,382	19,783	19,639
		cálculo	39,684	40,190	36,362	30,965
HARBOUR (704x576)	1500	*preparação*	72,020	41,181	47,309	47,026
		cálculo	67,443	67,638	52,911	45,479
	2000	*preparação*	68,297	40,730	47,701	46,940
		cálculo	61,988	65,378	51,207	43,907
	2500	*preparação*	72,917	40,946	46,075	45,947
		cálculo	62,378	63,649	50,271	43,280
	3000	*preparação*	72,867	40,904	46,127	46,286
		cálculo	60,529	62,981	49,792	42,714
ICE (704x576)	1500	*preparação*	69,698	41,247	49,357	47,671
		cálculo	99,871	97,512	85,612	73,341
	2000	*preparação*	69,528	40,868	48,897	48,710
		cálculo	85,761	88,471	74,307	63,472
	2500	*preparação*	74,423	40,841	46,216	46,813
		cálculo	78,829	81,354	67,757	57,993
	3000	*preparação*	74,052	40,044	46,044	45,777
		cálculo	73,804	76,828	63,656	54,479

A partir desses dados, podem-se calcular separadamente os ganhos referentes à preparação de estruturas e ao cálculo de movimento (tabela 5.3).

De forma similar à seção 5.3.1, os ganhos foram calculados dividindo-se os tempos de cada versão (multi-quadro, paralela normal e paralela otimizada) pelo tempo da versão básica.

Tabela 5.3: Ganho em função de preparação de estruturas e cálculo de movimento.

Sequência	Tempo gasto com	Ganho em relação à versão básica (%) Multi-quadro	Paralela normal	Paralela otimizada
CITY (704x576)	*preparação*	75,40	52,06	51,96
	cálculo	0,88	21,44	41,59
CREW (704x576)	*preparação*	74,51	52,34	52,27
	cálculo	**-2,99**	16,32	35,73
HARBOUR (704x576)	*preparação*	73,87	51,60	51,97
	cálculo	**-2,72**	16,32	35,81
ICE (704x576)	*preparação*	73,53	51,29	51,67
	cálculo	**-2,43**	16,22	35,76
MÉDIA	*preparação*	74,33	51,82	51,97
	cálculo	**-1,81**	17,57	37,22

Os resultados apresentados até este ponto mostram que os ganhos médios relativos ao paralelismo obtido com os SPEs são de 17,6% e 37,2%, referindo-se respectivamente às versões paralela normal (sem o uso de instruções SIMD) e paralela otimizada (com o uso de instruções SIMD), em relação à versão básica.

5.3.3 Estimativa de movimento por subquadro

A princípio, por se trabalhar com um algoritmo com 6 unidades de execução independentes e não correlacionadas, se imaginaria um ganho de cerca de seis. Entretanto, como o paralelismo é obtido por meio da utilização dos SPEs, que possuem arquiteturas funcionais bem distintas à de um PPE, uma relação direta de comparação não é válida.

Visando-se a estimar a perda de desempenho, no cálculo de cada subquadro, associada à execução do algoritmo de estimativa nos SPEs, novos testes foram realizados, obtendo-se uma análise ainda mais detalhada dos tempos de execução.

Estes testes consideram os tempos médios de cálculo de todos os vetores de movimento dentro de cada subquadro, os quais são apresentados de forma resumida na tabela 5.4.

Tabela 5.4: Tempo médio de estimativa de movimento por subquadro.

Sequência	Taxa (Kbps)	Tempo médio por subquadro (milhares de ticks)		
		Multi-quadro	Paralela normal	Paralela otimizada
CITY (704x576)	1500	47,421	192,474	165,263
	2000	46,665	186,535	160,469
	2500	46,219	185,843	159,679
	3000	45,650	181,588	156,227
CREW (704x576)	1500	70,576	313,326	265,643
	2000	67,494	298,788	253,367
	2500	65,555	290,470	246,466
	3000	64,392	287,730	244,202
HARBOUR (704x576)	1500	46,440	186,297	158,806
	2000	44,889	179,618	152,933
	2500	43,701	176,907	150,833
	3000	43,243	173,421	148,432
ICE (704x576)	1500	66,957	297,080	253,070
	2000	60,750	263,404	223,259
	2500	55,862	239,729	204,177
	3000	52,753	226,469	192,965
MÉDIA TOTAL		54,285	229,980	195,987

A partir desses dados, é possível calcular a relação média do tempo de execução dos códigos executando nos SPEs em relação ao PPE (tabela 5.5). Essa relação foi calculada dividindo-se os tempos das duas versões paralelas (normal e otimizada) pelo tempo da versão multi-quadro.

Tabela 5.5: Estimativa de perda associada ao processador.

Sequência	Taxa (Kbps)	Tempo relativo à versão multi-quadro	
		Paralela normal	Paralela otimizada
CITY (704x576)	1500	4,06	3,49
	2000	4,00	3,44
	2500	4,02	3,45
	3000	3,98	3,42
CREW (704x576)	1500	4,44	3,76
	2000	4,43	3,75
	2500	4,43	3,76
	3000	4,47	3,79
HARBOUR (704x576)	1500	4,01	3,42
	2000	4,00	3,41
	2500	4,05	3,45
	3000	4,01	3,43
ICE (704x576)	1500	4,44	3,78
	2000	4,34	3,68
	2500	4,29	3,66
	3000	4,29	3,66
MÉDIA TOTAL		4,20	3,58

Comparando-se os tempos de execução da versão que roda no PPE com os das versões que rodam nos SPEs, percebe-se uma perda de desempenho média de 3,6 (versão com SIMD) a 4,2 (versão sem SIMD) vezes do SPE em relação ao mesmo código rodando no PPE.

5.4 Qualidade

A divisão dos quadros em subquadros fez com que fosse impossibilitado o uso de alguns preditores no cálculo de vetores de movimento (blocos localizados nas bordas de cada subquadro).

Essa ausência de preditores leva, algumas vezes, a que o algoritmo de estimativa de movimento adotado acabe encontrando um SAD mínimo local diferente do que encontraria se considerasse todos os preditores.

O impacto desse erro é refletido nos resultados de qualidade (em PSNR) obtidos, conforme pode ser observado na tabela 5.6.

Tabela 5.6: Impacto da omissão de alguns preditores na qualidade final

Sequência	Taxa (Kbps)	PSNR-Y (dB)	
		Versão básica	Versões paralelas
CITY (704x576)	1500	31,012	30,999
	2000	31,745	31,748
	2500	32,412	32,396
	3000	33,030	32,984
CREW (704x576)	1500	36,400	36,387
	2000	37,205	37,200
	2500	37,837	37,809
	3000	38,333	38,336
HARBOUR (704x576)	1500	29,821	29,812
	2000	30,850	30,853
	2500	31,693	31,673
	3000	32,435	32,420
ICE (704x576)	1500	40,484	40,464
	2000	41,627	41,596
	2500	42,380	42,371
	3000	43,035	43,034

A tabela mostra que a diferença de PSNR associada à desconsideração de alguns preditores nas bordas dos subquadros é de no máximo 0,046dB, o que pode ser considerado pouco representativo. Na prática, como o olho humano não é sensível a variações dessa proporção, não é possível perceber visualmente a diferença na qualidade final da imagem.

5.5 Considerações finais sobre os experimentos

Os resultados dos ensaios mostram que a etapa de cálculo de movimento em subquadros, na versão paralela, chegou a executar 37,2% mais rápido que a versão básica. Para alcançar esse ganho de desempenho, foram utilizadas seis unidades de execução paralelas (SPEs).

Com base nos dados obtidos, pode-se determinar uma estimativa do ganho que seria obtido nesta etapa, caso a execução nos SPEs levasse o mesmo tempo que a execução no PPE. A tabela 5.7 mostra essa estimativa.

Tabela 5.7: Estimativa de ganho desconsiderando perda associada ao processador.

	Perda média associada ao processador	Ganho médio por subquadro	Ganho estimado
Paralela normal	3,584	1,565	5,609
Paralela otimizada	4,203	1,342	5,641

Uma das estratégias utilizadas para possibilitar a paralelização foi a de desconsiderar os preditores que estivessem em subquadros vizinhos. Os resultados apresentados neste capítulo mostram que o impacto disso na qualidade final de imagem não é perceptível.

6 CONCLUSÃO

Neste trabalho foi estudada uma solução paralela para o algoritmo de estimativa de movimento em vídeos. Para sua implementação prática, utilizou-se o processador CBE, por possuir uma arquitetura na qual se pode explorar um alto grau de paralelismo, utilizando um microprocessador central (PPE) que pode rodar em paralelo com até oito processadores DSP (SPEs). Outra razão para a escolha do CBE como plataforma de desenvolvimento foi o conjunto de instruções SIMD das SPEs, que operam sobre vetores de 128 bits, podendo ser aproveitadas para cálculos de computação intensiva, como os realizados no algoritmo de estimativa de movimento.

O processo de codificação de vídeo é essencial para que se consiga armazenar e transmitir vídeos digitais. Dentre os algoritmos envolvidos na codificação, o de estimativa de movimento pode ser destacado como um dos mais críticos, devido ao seu alto custo computacional. O objetivo desse algoritmo é ajudar a aproveitar redundâncias temporais nas imagens, reduzindo a taxa de bits de saída na codificação.

A paralelização desse algoritmo foi realizada a partir da segmentação dos quadros em imagens menores, ou subquadros. Os cálculos de movimento associados aos blocos de cada subquadro foram realizados paralelamente em SPEs distintas. Para que isso fosse possível, foi preciso utilizar uma estratégia de desconsiderar preditores de posição inicial que estivessem em subquadros vizinhos, o que afetou a busca de vetores nas bordas dos subquadros. Entretanto, o impacto causado na qualidade final de imagem não é perceptível ao olho humano, e os ganhos obtidos com a paralelização ainda se justificam.

A plataforma de testes escolhida, o PlayStation 3, somente permite a utilização de seis SPEs para programas de usuário. Apesar dessa limitação, conseguiu-se alcançar com o algoritmo paralelo de estimativa de movimento um ganho de desempenho médio de 33% em relação à sua versão básica (não paralela). Analisando-se de forma isolada a etapa do algoritmo que foi efetivamente paralelizada, o ganho obtido foi de 17%. Aprimorando-se o módulo de cálculo de SAD, com base no uso de instruções SIMD, esses ganhos chegaram, respectivamente, a 45% e 37%.

Entretanto, esses ganhos ainda são limitado devido à execução, nos SPEs, de código não inteiramente otimizado para esse tipo de processador. Mesmo com o uso de instruções SIMD para o cálculo de SAD, o algoritmo de busca por blocos similares dentro de um subquadro ainda roda 3,6 mais rápido no PPE do que nos SPEs. Em um trabalho futuro, podem ser estudadas maneiras de se adaptar a forma como os SPEs tratam as manipulações de memória, de forma a aproveitar ainda mais as vantagens de operações vetoriais. Além disso, pode-se fazer um estudo do impacto causado pelo número de subquadros utilizados, resolução do vídeo e número de SPEs utilizados.

Desenvolver uma aplicação que explore os recursos do CBE não é uma tarefa simples. Durante a implementação, é necessário que se conheça e considere uma grande quantidade de detalhes relativos à arquitetura do CBE. Tanto os alinhamentos de memória como as tranferências DMA precisam ser feitas explicitamente no código do programa. Além disso, o limite de 256KB da memória local dos SPEs não pode ser ultrapassado, e cabe ao programador tomar esse tipo de cuidado.

REFERÊNCIAS

AMORIM, A. C. O. **A Computação Paralela e o Processador Cell**. 2005. 18 f. Trabalho Individual (Pós-Graduação em Sistemas Eletrônicos) - Escola Politécnica, USP, São Paulo.

GECKLE, W. **Lecture Notes 9**: Motion Estimation. 2000. Disponível em: <http://www.apl.jhu.edu/Notes/Geckle/525759/lecture9.pdf>. Acesso em: jun, 2010.

GOREN, A. Multimedia needs multiprocessor SoCs. **EE Times Magazine**, July 2003. Disponível em: <http://www.eetimes.com/story/OEG20030702S0054>. Acesso em: jun, 2010.

GREHS, I. A. **An LDPC Codes Simulator on the Cell Broadband Engine**. 2009. 50 f. Projeto de Diplomação (Bacharelado em Engenharia de Computação) - Instituto de Informática, UFRGS, Porto Alegre.

GSCHWIND, M. et. al. Synergistic Processing in Cell's Multicore Architecture. **IEEE Micro**. [S.l.], v.26, n.2, p. 10-24, mar. 2006.

HÜSEMANN, R. et. al. Optimized SAD Calculation Algorithm for Cell Processor. In: SIMPÓSIO BRASILEIRO DE SISTEMAS MULTIMÍDIA E WEB, WEBMEDIA, 2008. **Anais...** Vila Velha: [s.n.], 2008.

HÜSEMANN, R. et. al. **Proposal of an Improved Motion Estimation Module for SVC**. In: Symposium On Applied Computing, SAC, 2010.

IBM. **Cell Broadband Engine Programming Handbook**: Including the PowerXCell 8i Processor. [S.l.: s.n.], 2008.

IBM. **Programming the Cell Broadband Engine Architecture**: Examples and Best Practices. [S.l.: s.n.], 2008.

JACK, K. **Video Demystified**: A Handbook for the Digital Engineer. 5th ed. [S.l]: Elsevier Science & Technology Books, 2007.

RICHARDSON, I. E. G. **H.264 and MPEG-4 Video Compression**: Video Coding for Next-Generation Multimedia. [S.l]: Wiley & Sons, 2003.

SÜHRING, K. **H.264/AVC JM Reference Software**. 2009. Disponível em: <http://iphome.hhi.de/suehring/tml/download/>. Acesso em: jun, 2010.

TOURAPIS, A. M.; AU, O. C.; LIOU, M. L. Predictive Motion Vector Field Adaptive Search Technique (PMVFAST): Enhancing Block Based Motion Estimation. In:

VISUAL COMMUNICATIONS AND IMAGE PROCESSING, VCIP, 2001. **Proceedings...** San Jose, CA: [s.n.], 2001. p.883-892.

WYGANOWSKI, M. **Classification Algorithms on the Cell Processor**. 2008. 166 p. Thesis (Master of Science in Computer Engineering) – Department of Computer Engineering, Kate Gleason College of Engineering, RIT, Rochester.