Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

Guilherme Mota Cavalcanti de Albuquerque Cox

# Implementação de Visualização de Dados Tridimensionais de Malhas Irregulares no Processador Cell Broadband Engine

Rio de Janeiro
2009

# Implementação de Visualização de Dados Tridimensionais de Malhas Irregulares no Processador Cell Broadband Engine

Guilherme Mota Cavalcanti de Albuquerque Cox

Dissertação apresentada, como requisito para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia de Computação, da Faculdade de Engenharia da Universidade do Estado do Rio de Janeiro. Área de concentração: Geomática.

Orientador: Cristiana Barbosa Bentes
Co-orientador: Ricardo Cordeiro de Farias

Rio de Janeiro
2009

UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO
UERJ

CATALOGAÇÃO NA FONTE
UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

Cox, Guilherme

Implementação de Visualização de Dados Tridimensionais de Malhas Irregulares no Processador Cell Broadband Engine/ Guilherme Mota Cavalcanti de Albuquerque Cox– 2009

79 fls : i1

Orientador: Cristiana Barbosa Bentes

Co-orientador: Ricardo Cordeiro de Farias

Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

Bibliografia: 71–78

1. Visualização Volumétrica - Teses. 2. Renderização Paralela. I. Bentes, Cristiana B. e II. Farias, Ricardo C.

Autorizo, apenas para fins acadêmicos e científicos, a reprodução total ou parcial desta dissertação.

Assinatura                    Data

Guilherme Mota Cavalcanti de Albuquerque Cox

**Implementação de Visualização de Dados Tridimensionais de Malhas Irregulares no Processador Cell Broadband Engine**

Dissertação apresentada, como requisito para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia de Computação, da Faculdade de Engenharia da Universidade do Estado do Rio de Janeiro. Área de concentração: Geomática.

Aprovado em: ______________________________

Banca Examinadora:

______________________________________________
Prof. D.Sc. Cristiana Barbosa Bentes(Orientador)
Faculdade de Engenharia – UERJ

______________________________________________
Prof. Ph.D. Ricardo Cordeiro de Farias(Co-orientador)
COPPE/Sistemas – UFRJ

______________________________________________
Prof. D.Sc. Guilherme Lúcio Abelha Mota
Faculdade de Engenharia – UERJ

______________________________________________
Prof. D.Sc. Esteban Walter Gonzalez Clua
Instituto de Computação – UFF

Rio de Janeiro
2009

# RESUMO

Cox, Guilherme. Implementação de Visualização de Dados Tridimensionais de Malhas Irregulares no Processador Cell Broadband Engine.

Dissertação (Mestrado em Engenharia de Computação) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2009.

A renderização de volume direta tornou-se uma técnica popular para visualização volumétrica de dados extraídos de fontes como simulações científicas, funções analíticas, scanners médicos, entre outras. Algoritmos de renderização de volume, como o *raycasting*, produzem imagens de alta qualidade. O seu uso, contudo, é limitado devido à alta demanda de processamento computacional e o alto uso de memória. Nesse trabalho, propomos uma nova implementação do algoritmo de *raycasting* que aproveita a arquitetura altamente paralela do processador Cell Broadband Engine, com seus 9 núcleos heterogêneos, que permitem renderização eficiente em malhas irregulares de dados. O poder computacional do processador Cell BE demanda um modelo de programação diferente. Aplicações precisam ser reescritas para explorar o potencial completo do processador Cell, que requer o uso de multithreading e código vetorizado. Em nossa abordagem, enfrentamos esse problema distribuindo a computação de cada raio incidente nas faces visíveis do volume entre os núcleos do processador, e vetorizando as operações da integral de iluminação em cada um. Os resultados experimentais mostram que podemos obter bons *speedups* reduzindo o tempo total de renderização de forma significativa.



Palavras-chave: Visualização científica, Computação de alto desempenho

# ABSTRACT

Direct volume rendering has become a popular technique for visualizing volumetric data from sources such as scientific simulations, analytic functions, and medical scanners, among others. Volume rendering algorithms, such as raycasting, can produce high-quality images, however, the use of raycasting has been limited due to its high demands on computational power and memory bandwidth. In this paper, we propose a new implementation of the raycasting algorithm that takes advantage of the highly parallel architecture of the Cell Broadband Engine processor, with 9 heterogeneous cores, in order to allow interactive raycasting of irregular datasets. All the computational power of the Cell BE processor, though, comes at the cost of a different programming model. Applications need to be rewritten in order to explore the full potential of the Cell processor, which requires using multithreading and vectorized code. In our approach, we tackle this problem by distributing ray computations using the visible faces, and vectorizing the lighting integral operations inside each core. Our experimental results show that we can obtain good speedups reducing the overall rendering time significantly.

Keywords: Raycasting, Parallel Programming, High Performance Systems

# Sumário

**Lista de Figuras** **4**

**Lista de Tabelas** **5**

**1 Introdução** **6**

**2 Trabalhos Relacionados** **10**

2.1 Aceleração da renderização de volume . . . . . . . . . . . . . . . . . . 10

2.1.1 Renderização Paralela em *Clusters* . . . . . . . . . . . . . . . . 11

2.1.2 Renderização em GPUs . . . . . . . . . . . . . . . . . . . . . . 12

2.1.3 Explorando Arquiteturas SIMD . . . . . . . . . . . . . . . . . . 13

2.2 Explorando o processador Cell BE . . . . . . . . . . . . . . . . . . . . . 14

**3 Visualização Volumétrica** **15**

3.1 Conceitos Básicos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

3.2 Técnicas de Renderização . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.3 Representação de Dados . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.4 Algoritmos de Renderização Volumétrica . . . . . . . . . . . . . . . . . 18

3.4.1 Projeção de Células . . . . . . . . . . . . . . . . . . . . . . . . 18

3.4.2 *Raycasting* . . . . . . . . . . 18

3.5 VF-Ray: *Visible Faces Raycasting* . . . . . . . . . . 19

**4 O Processador Cell Broadband Engine** **22**

4.1 A arquitetura do processador Cell BE . . . . . . . . . . 22

4.1.1 O *Power Processing Element* (PPE) . . . . . . . . . . 23

4.1.2 O *Synergistic Processing Element* (SPE) . . . . . . . . . . 24

4.1.3 O *Element Interconnection Bus* (EIB) . . . . . . . . . . 25

4.1.4 PPE $\times$ SPE . . . . . . . . . . 25

4.2 Programando para o processador Cell BE . . . . . . . . . . 26

4.2.1 Explorando as habilidades SIMD . . . . . . . . . . 27

4.2.2 Paralelizando para o SPE . . . . . . . . . . 28

4.2.3 Transferências DMA . . . . . . . . . . 30

4.2.4 Transferência eficiente DMA e sobreposição de computação . . 31

4.2.5 Programando para o PPE . . . . . . . . . . 33

4.2.6 Programando para o SPE . . . . . . . . . . 36

4.2.7 Orientações básicas no desenvolvimento de código . . . . . . . 39

4.2.8 Dicas de implementação . . . . . . . . . . 39

4.3 Ambiente de desenvolvimento . . . . . . . . . . 42

4.4 SystemSim, O Simulador IBM do Cell BE . . . . . . . . . . 43

**5 Implementando o VF-Ray no Cell BE** **45**

5.1 Desafios . . . . . . . . . . 45

5.2 O Algoritmo Cell-VF-Ray . . . . . . . . . . 47

5.2.1 Paralelizando para o SPE . . . . . . . . . . 47

5.2.2 Explorando as facilidades SIMD . . . . . . . . . . 49

5.2.3 Transferências DMA . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.4 Evitando Desvios . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.3 Algoritmo Cell-VF-Ray-FB . . . . . . . . . . . . . . . . . . . . . . . . 52

**6 Resultados Experimentais 55**
6.1 Ambientes de Teste . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
6.1.1 Configuração do Ambiente . . . . . . . . . . . . . . . . . . . . . 56
6.2 Massas de Dados . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.3 Cell-VF-Ray . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.3.1 Resultados da Simulação . . . . . . . . . . . . . . . . . . . . . . 60
6.4 Cell-VF-Ray-FB . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.5 Comparação com a plataforma Intel . . . . . . . . . . . . . . . . . . . 64
6.6 Discussão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

**7 Conclusões 70**

**Referências Bibliográficas 72**

# Lista de Figuras


3.1 Esquema do algoritmo de *raycasting*. . . . . . . . . . . . . . . . . . . . . 19

4.1 Arquitetura do processador Cell BE. . . . . . . . . . . . . . . . . . . . . 23

4.2 Quatro operações de adição em paralelo, em cada elemento dos dois vetores. 28

4.3 Exemplo da técnica de *double buffering*. . . . . . . . . . . . . . . . . . . 32

4.4 Esquema dos tipos de dados e dos *slots* preferidos. . . . . . . . . . . . 38

6.1 Playstation 3: Arquitetura Cell BE. . . . . . . . . . . . . . . . . . . . . . 57

6.2 SPX. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

6.3 Blunt Fin. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

6.4 Oxigen Post. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

6.5 Delta Wing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

6.6 Fighter. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

6.7 F-117. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

6.8 Torso. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

6.9 Tempo de execução para todos *datasets* - sem *unroll* de *loops*. . . . . . 69

# Lista de Tabelas


4.1 Diferenças na arquitetura do PPE v SPE [25]. . . . . . . . . . . . . . . . 33

6.1 Descrição das massas de dados. . . . . . . . . . . . . . . . . . . . . . . . 58

6.2 Ganho de desempenho devido a otimização de *unroll* de *loops*. . . . . . 59

6.3 *Speedups* de 1 até 6 SPEs do Cell-VF-Ray. . . . . . . . . . . . . . . . . . 59

6.4 Tempo de execução por tamanho da imagem do Cell-VF-Ray. . . . . . 60

6.5 *Speedups* de 1 até 6 SPEs do Cell-VF-Ray-FB. . . . . . . . . . . . . . 63

6.6 Tempo de execução por tamanho da imagem do Cell-VF-Ray-FB. . . . 63

6.7 Comparação de *speedups* entre Cell-VF-Ray e Cell-VF-Ray-FB. . . . . 64

6.8 Comparação de tempos de execução, Cell-VF-Ray × VF-Ray (Intel). . 65

6.9 Comparação de tempos de execução, Cell-VF-Ray-FB × VF-Ray (Intel). 65

# Capítulo 1

# Introdução

Interpretar o mundo que nos cerca através de imagens é uma atividade que nós, seres humanos, temos nos empenhado por toda nossa história. A visualização científica desempenha um importante papel no processo científico, pois simulações, experimentos e coletas de dados abrangem um enorme e contínuo acumulo de dados. A visualização ajuda cientistas a obter informações sobre as estruturas dos dados, as relações e anomalias escondidas dentro dos conjuntos de dados reais ou simulados. Áreas de aplicação como medicina, geologia, biologia, química, dinâmica dos fluidos, ciência molecular ou de proteção ambiental, dependem cada vez mais da visualização científica como uma forma eficaz de obter a compreensão intuitiva de um problema.

Nos sistemas de informação geográfica, em particular, a visualização científica tem sido cada vez mais importante para melhorar, por exemplo, a modelagem de oceanos [14, 20], o monitoramento da poluição atmosférica [13], a visualização de dados meteorológicos [53], a modelagem de terrenos [47], e a compreensão de alguns fenômenos naturais como os ciclones tropicais [58].

Existem técnicas diferentes para a visualização de dados tridimensionais. Basica-

mente, podem ser diferenciadas entre métodos indiretos e diretos. Os métodos indiretos baseiam-se na reconstrução de dados poligonais, que podem ser renderizados como dados de isosuperfície. Neste método, apenas parte do conjunto de dados volumétricos contribui diretamente para a imagem resultante. Nos métodos diretos, por outro lado, todos os dados têm potencial de contribuir para a imagem resultante. O objetivo principal dos métodos diretos, chamados de métodos de renderização de volume, é mapear um conjunto de informações, definidas através de uma malha volumétrica, para valores de cor e de opacidade na imagem final. Renderização de volume tem a vantagem de visualizar o conjunto de dados completo, expondo suas estruturas internas, produzindo imagens de alta qualidade, porém, devido à quantidade excessiva de amostras e a complexidade das operações realizadas, ela exige um poder computacional elevado.

Além disso, dependendo do tipo de malha usada para representar os dados volumétricos, os requisitos de memória podem também ser um obstáculo para os algoritmos de renderização de volume. As malhas irregulares, com células de tetraedros ou hexaedros, têm os vértices em posições arbitrárias. Malhas regulares, por outro lado, têm células cúbicas, onde as distâncias entre pontos adjacentes permanecem constantes ao longo dos dados. No entanto, malhas regulares geralmente não são apropriadas para representação de diferentes campos escalares, já que o tamanho da célula precisa ser pequeno o suficiente para modelar a menor caracteristica do campo [10]. Por essa razão, representações de volumes em malhas irregulares estão se tornando importantes na pesquisa de visualização volumétrica, já que elas podem representar mais detalhes apenas aonde é realmente necessário, se encaixando naturalmente na forma que determinados sistemas físicos são modelados. Entretanto, devido a falta de regularidade nas estruturas dessas malhas, as coordenadas dos vértices precisam ser representadas explicitamente, assim como a conectividade entre eles. Como resultado, malhas irregulares de volume

necessitam de espaço de armazenamento relativamente grande durante o processo de renderização.

Portanto, os principais desafios na concepção dos algoritmos de renderização de volume em alto desempenho para malhas irregulares são: (i) o custo computacional necessário para atravessar os dados com objetivo de computar a cor e a opacidade de cada pixel na tela, e (ii) a memória necessária para manter as informações de conectividade dos dados.

Esforços de pesquisa significativos vêm tentando enfrentar esses desafios utilizando processadores gráficos (GPU) [16, 37, 39, 44] ou processamento paralelo em *clusters* de computadores [62, 33, 52, 35]. Esses estudos tiram proveito das contínuas melhorias na performance da CPU e da GPU.

Apesar dos grandes avanços alcançados por esses dois enfoques, há alguns incovenientes em suas implementações. Na GPU um programador tem que enfrentar capacidade de memória limitada e a alta latência GPU-CPU de transfêrencia de dados. Enquanto que a implementação em *clusters* necessita sintonia fina na implementação para resolver questões de distribuição de dados, distribuição de carga, composição de imagem, e problemas de comunicação.

Assim, neste trabalho, propomos uma forma alternativa para renderização de volume em malhas irregulares, diferente do uso de GPUs ou *clusters*: exploramos o poder do processador Cell Broadband Engine. O processador Cell BE é uma nova arquitetura *multicore* heterogênea desenvolvida pela IBM, Sony, e Toshiba [29]. Ele é usado no console de video-game Playstation3 (PS3), e é também no supercomputador mais rápido do mundo atualmente, o *RoadRunner*, como anunciado no último Top500 [15]. No entanto, todo o poder computacional do processador Cell BE acompanha um modelo de programação diferente. Aplicações precisam ser reescritas para executarem efetiva-

mente no processador Cell BE. O desempenho geral da aplicação depende altamente da utilização eficaz dos recursos do Cell BE, que são em grande parte deixados sob responsabilidade do programador.

Nossa abordagem aqui é implementar no Cell BE, um algoritmo de *raycasting* com uso eficiente de memória para renderização de volume em malhas irregulares, que ataque dois desafios principais: (i) conformidade com o modelo de memória do Cell BE, e (ii) explorararação da programação vetorial SIMD (*Single Instruction Multiple Data*). Propomos um novo algoritmo paralelo que distribui a computação dos raios que atravessam as faces visíveis entre os *cores*, e vetorizamos as operações da integral de iluminação dentro de cada *core*. Nossos resultados experimentais mostram que podemos obter bons *speedups* com a exploração da grande capacidade paralela do processador.

O restante dessa dissertação está organizado da seguinte forma. No próximo capítulo 2, revisaremos trabalhos relacionados a algoritmos de renderização de volume e ao uso do processador Cell BE. No capítulo 3, descrevemos a visualização volumétrica e o algoritmo de *raycasting*, usado como base na nossa implementação, chamado VF-Ray. No capítulo 4, descrevemos a arquitetura do processador Cell BE. No capítulo 5, mostramos todas mudanças no código que foram feitas para explorar o poder computacional do processador Cell BE. No capítulo 6, reportamos nossos resultados experimentais e no capítulo 7 apresentamos nossas conclusões e planos para pesquisas futuras.

# Capítulo 2

# Trabalhos Relacionados

Até onde sabemos, essa é a primeira implementação do algoritmo de *raycasting* usando dados de malhas irregulares no processador Cell BE. Assim, neste capítulo não iremos comparar nosso algoritmo com outras implementações, mas faremos um breve resumo da literatura em termos de: aceleração da renderização de volume, e programação para o processador Cell BE.

## 2.1 Aceleração da renderização de volume

Nos últimos anos, tem crescido a literatura sobre algoritmos e técnicas de aceleração da renderização de volume, em particular com foco nas implementações em GPU e arquiteturas de *clusters*. Estudos concentram-se em implementações eficientes e otimizadas dos algoritmos de renderização de volume e suas estruturas de dados, com objetivo de explorar os recursos da GPU, evitar gargalos de execução em *clusters* e explorar a capacidade de processamento SIMD dos processadores.

### 2.1.1 Renderização Paralela em *Clusters*

Vários algoritmos de renderização paralela baseados em *clusters* foram propostos ao longo dos anos. As três abordagens básicas em processamento paralelo são: *sort-first*, *sort-middle*, e *sort-last*, como definido por Molar*et al.* [41].

Na renderização paralela *sort-first*, o espaço da tela (imagem final) é dividido em retângulos chamados *tiles* e cada processador recebe um conjunto de *tiles* para processar. Normalmente, essa abordagem apresenta menor comunicação entre as unidades de processamento, porém ela é muito suscetível ao desbalanceamento de carga. Existem vários trabalhos que tratam o problema do balanceamento de carga de algoritmos *sort-first*. Samanta *et al.* [52] propuseram algumas técnicas interessantes, mas com escopo na renderização de superfície. Para renderização de volume, Coelho *et al.* [9] e Farias *et al.* [17] apresentaram eficientes algoritmos de roubo de trabalho para *clusters*. Mueller explorou em [42, 43] a coerência quadro-a-quadro a fim de permitir um bom particionamento da tela. O trabalho de Abraham *et al.* [1] também concentrou-se na partição da tela. Eles propuseram redimensionamento do tile a fim de promover a mesma quantidade de trabalho distribuído. Num trabalho recente, Lanbronici *et al.* [32] propuseram um outro algoritmo de partição da tela, que é adaptável e baseado em uma divisão quad-tree. Samanta *et al.* desenvolveram em [48] uma abordagem híbrida *sort-first*/*sort-last* melhor escalável em relação ao número de unidades de processamento e a resolução da tela. Já o trabalho de Correa *et al.* [11], focou na renderização de grandes modelos que não cabem na memória principal, numa abordagem *out-of-core*.

Na renderização paralela *sort-last* cada nó de renderização é responsável por renderizar parte do objeto da cena. Ela tem sido amplamente usada em diferentes trabalhos. Os trabalhos [35, 44, 61] trataram a distribuição de carga dividindo o volume em blocos,

e redistribuindo blocos para nós menos sobrecarregados. Outra importante questão nos algoritmos *sort-last* é o estágio final de composição da imagem. Essa etapa tem grande chance de se tornar concorrida por todos nós e afunilar o paralelismo do processo, já que demanda grande quantidade de troca de mensagens entre os nós. Os trabalhos de Yu *et al.* [62] e Lee *et al.* [33] focaram em reduzir esse excesso de troca de mensagens. Yu *et al.* apresentaram um novo algoritmo de composição de imagem, chamado 2-3 *swap*, que oferece benefícios no envio direito, e na troca binária propostos em [34]. Lee *et al.* mostraram um método de pipeline paralelo que evita contenção de links. Childs *et al.* [7] focaram na escabilidade do paralelismo e propuseram uma abordagem híbrida cujo paralelismo é baseado na divisão dos dados de entrada (*input*) e dos pixels da imagem de saída (*output*).

### 2.1.2 Renderização em GPUs

Uma segunda técnica de aceleração da renderização de volume é baseada em explorar o *hardware* de uma GPU. Atualmente, a GPU vendida nas placas de vídeo evoluiu para um poderoso e flexível processador. Seu paralelismo intrínsico é muito conveniente para renderização de volume. A eficiente combinação de otimizações algorítimicas na GPU permitiram que o desempenho da renderização de volume seja levado para um nível interativo. Em [59], Weiler *et al.* implementaram um algoritmo de *raycasting* baseado em *hardware*, chamado HARC, que foi posteriormente extendido por Espinha e Celes [16]. Bernardon *et al.* [5] também propuseram um algoritmo de *raycasting* baseado em *hardware*, que renderiza dados em malhas irregulares não convexas. Ruijters *et al.* [50] investigaram esquemas de otimização para algoritmos de *raycasting* em GPU usando a combinação de blocos de processamento, *kd-trees*, e terminação antecipada do

raio. Muitas tentativas têm sido feitas para lidar com problema do limite de memória da GPU. Weiler *el al.* propuseram em [60] o uso de métodos de compactação de dados, tiras de tetraedros, para reduzir necessidade de memória. Fout e Ma [19] implementaram uma compressão volumétrica na qual a descompressão é acoplada à renderização. Maximo *et al.* [39] implementaram um novo esquema na ordenação dos dados da face em um algoritmo de *raycasting* baseado em GPU.

Recentemente alguns trabalhos combinam os benefícios tradicionais da computação paralela com o alto desempenho oferecido pelas técnicas baseadas em GPU, desenvolvendo algoritmos de renderização de volume para *clusters* de GPU. Strengert *et al.* propuseram uma abordagem adaptada baseada em texturas em [55]. Muller *et al.* apresentaram uma proposta *sort-last* de algoritmo de *raycasting* paralelo em [44]. Allard and Raffin mostraram, em [3], o FlowVR, um framework baseado em sombras. Marchesin *et al.* [36] propuseram um algoritmo *sort-last* baseado em múltiplas GPUs em uma única CPU. Lee and Newman em [33] propuseram uma técnica de alto desempenho em correção de opacidade para *raycasting* em um *clusters* de GPUs. WireGL [21] e seu sucessor Chromium [22] unificaram o poder de renderização de uma coleção de aceleradores gráficos em nós de um *cluster*.

### 2.1.3 Explorando Arquiteturas SIMD

O trabalho de Meibner *et al.* [40] mostrou uma implementação do algoritmo de *raycasting* paralelo para projeções ortogonais numa arquitetura de *single-chip* SIMD, o FUZION *chip*. Processamento simultâneo de raios é programado de maneira que os acessos à memória dos elementos individuais de processamento podem ser detectados pelo canal de controle. Os trabalhos de Adinetz *et al.* [2] e Wald *et al.* [57] exploram o

SSE *(Streaming SIMD Extension)* da arquitetura Intel para acelerar o processo de renderização. Eles focaram, contudo, em algoritmos *raytracing* que tratam apenas os dados de superfície.

## 2.2 Explorando o processador Cell BE

Sendo relativamente uma arquitetura nova, o potencial completo do processador Cell ainda está sendo explorado. Alguns trabalhos iniciais foram desenvolvidos pela IBM no campo de imagens médicas [51], computação FFT [8], e modelos moleculares [56]. Esses trabalhos forneceram resultados indicando *speedups* substanciais quando comparados com processadores convencionais.

Em termos da visualização de volume, o trabalho de Benthin *et al.* [4] e O'Conor *et al.* [45] exploraram a arquitetura do processador Cell para o algoritmo de *raytracing* para dados de superfície. A IBM também publicou uma implementação de *raycasting*, em que apenas as faces visíveis são avaliadas para gerar a imagem dos dados da superfície. O trabalho recente de Kim e Jaja [31] é o mais próximo ao nosso. Eles implementaram o algoritmo de *raycasting* em um processador Cell. Contudo, a abordagem deles usou apenas malhas regulares de dados.

# Capítulo 3

# Visualização Volumétrica

Neste capítulo, vamos apresentar alguns conceitos básicos sobre visualização volumétrica. Nosso foco é descrever as diferentes técnicas de renderização, a forma de representação de dados e os algoritmos de renderização volumétrica, com ênfase para o algoritmo de *raycasting* que é o foco deste trabalho. Por fim, descrevemos em mais detalhes o algoritmo de visualização utilizado como base para nossa implementação no processador Cell BE; VF-Ray.

## 3.1 Conceitos Básicos

A visualização de dados tridimensionais, também chamada de visualização volumétrica, é o processo no qual uma imagem bidimensional é gerada a partir de um conjunto de dados tridimensionais, permitindo a cientistas uma análise visual de seus experimentos, possibilitando uma melhor compreensão de processos de difícil observação.

O desenvolvimento de aplicações de visualização visa fornecer aos cientistas ferramentas que auxiliem nas mais variadas tarefas que requerem formas de analisar, exibir

e explorar a superfície e o interior de grandes volumes de dados (que podem variar com o tempo), para permitir que o usuário consiga identificar características significativas e obter os resultados desejados mais fácil e rapidamente.

Grandes volumes de dados tridimensionais podem ser obtidos por fontes como: dispositivos de captura como tomografia computadorizada ou ressonância magnética; simulações (elementos finitos, diferenças finitas); satélites de observação da Terra; bóias no oceano; ou técnicas de modelagem.

## 3.2 Técnicas de Renderização

As técnicas de visualização de volume podem ser classificadas em dois tipos: *surface rendering* (ou “visualização através de superfícies”) e *volume rendering* (ou “visualização direta de volume”) [30]. Técnicas de visualização através de superfícies envolvem a extração e a representação de uma isosuperfície que é posteriormente visualizada. Neste caso, apenas os dados da superfície do volume são representados na imagem. As grandes vantagens desta técnica, entretanto, são a velocidade para a geração e exibição da imagem final e o pouco espaço de armazenamento requerido. Representações deste tipo são apropriadas quando existem isosuperfícies bem definidas nos dados, mas não são eficientes quando o volume é composto por muitas microestruturas, tais como os tecidos em imagens médicas.

A segunda classe, volume rendering ou visualização direta de volume, consiste em representar o volume através de voxels 3D que são projetados diretamente em pixels 2D e armazenados como uma imagem. O seu principal objetivo é exibir as estruturas existentes no interior do volume de dados, permitindo a identificação e exploração de estruturas internas no volume. Como o volume é normalmente um "bloco de da-

dos", não se pode visualizar o seu interior, a menos que se assuma que é possível ver através de voxels transparentes. A renderização de volume permite uma análise mais detalhada dos dados quando comparada à renderização de superfície, dado que o interior dos dados também é considerado, o que é fundamental para diversas aplicações, como por exemplo, na localização de tumores no interior do corpo humano. A principal desvantagem da renderização volumétrica, entretanto, é o seu alto custo computacional. A computação de cada imagem requer milhões ou bilhões de operações inteiras e de ponto-flutuante.

## 3.3 Representação de Dados

Dados tridimensionais, obtidos sob qualquer tipo de captura, compreendem um conjunto de pontos no espaço tridimensional, onde a cada ponto estão associados propriedades ou fenômenos observados. Segundo Kaufman [30], dados volumétricos são tipicamente um conjunto $S$ de amostras ($x$, $y$, $z$, $v$), sendo $v$ o valor de alguma propriedade mensurável dos dados, como cor e densidade, e ($x$, $y$, $z$) a sua localização no espaço tridimensional. Por exemplo, no caso de dados coletados do oceano, essas propriedades podem ser sua temperatura, massa ou salinidade. Para que esses dados possam ser visualizados eles devem ser representados por uma estrutura geométrica renderizável. Esta estrutura geométrica é uma malha poliedral composta por um conjunto de células tridimensionais chamadas voxels.

Quando os voxels são cubos idênticos, a malha é dita regular. Malhas irregulares, por sua vez, são compostas de células poliedrais arbitrárias. Por esse motivo, elas são mais genéricas.

## 3.4 Algoritmos de Renderização Volumétrica

Na renderização de volume o dado é considerado como composto de um material semi-transparente, permitindo que a luz o atravesse. Muitas vezes, os dados presentes no interior de um volume contêm diversos materiais. Variações locais nas propriedades volumétricas, tais como absorção ou emissão de luz, são perdidas se o volume é reduzido a uma superfície. A renderização de volume permite que o cientista veja além da superfície.

Geralmente, a renderização de volume demanda recursos computacionais robustos. Por outro lado, gera imagens de excelente qualidade, já que todos os voxels são considerados. Dentre os algoritmos de renderização de volume mais importantes podemos destacar: *ray-casting* e projeção de células.

### 3.4.1 Projeção de Células

No algoritmo de projeção de células, as células, ou voxels, de uma malha são percorridos e suas faces são projetadas na tela. A projeção de uma face na tela corresponde ao cálculo da contribuição que a célula gera na cor final da imagem para cada pixel da tela cuja linha de direção de visão intersecta a célula. Diversos algoritmos de projeção de células foram propostos na literatura, dentre eles podemos destacar o algoritmo Zsweep [18].

### 3.4.2 *Raycasting*

A idéia básica por trás do algoritmo de *raycasting* é lançar raios de um ponto de observação através de cada pixel da imagem. Para volumes de dados irregulares, a medida que um raio caminha, ele entra no volume intersecionando um número de células, como

ilustrado na Figura 3.1. Cada par de interseções é usado para computar a contribuição da célula na cor e opacidade do pixel (raio). O raio termina quando ele deixa o volume ou quando ele atinge opacidade total.

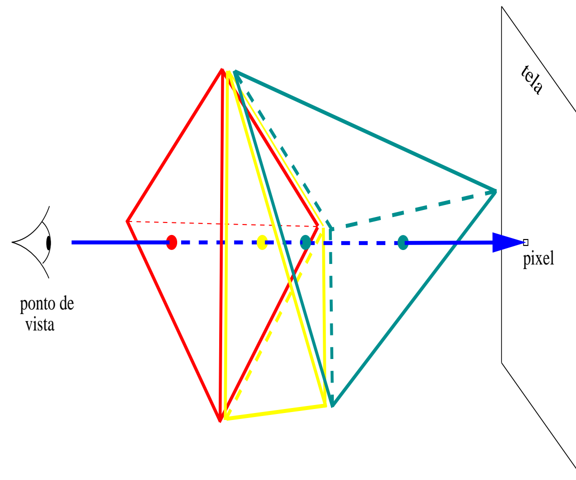


Figura 3.1: Esquema do algoritmo de *raycasting*.

A grande vantagem dos métodos de *raycasting* são: a computação de cada pixel é independente em relação a todos os outros pixels, e na renderização de malhas irregulares, o caminho do raio através da malha é guiado pela conectividade das células, evitando a necessidade de ordená-las. A desvantagem é o alto consumo de memória.

## 3.5 VF-Ray: *Visible Faces Raycasting*

Baseamos nossa implementação de *raycasting* para o Cell no algoritmo de *raycasting* com uso de memória eficiente para malhas irregulares, chamado de *Visible Faces Raycasting* (*VF-Ray*) [49]. Esse algoritmo usa uma estrutura de dados mais compacta e não redundante proporcionando ganhos consistentes e significativos de consumo de memória.

O foco principal do VF-Ray é reduzir drasticamente o uso de memória para poder renderizar conjuntos de dados extremamente grandes, baseado no fato de que a decisão de como armazenar as informações sobre as faces de cada célula é a chave para o consumo de memória. Essas informações são guardadas numa estrutura de dados da face, que inclui sua geometria (geralmente três vértices), e seus coeficientes (da equação do plano definido pelos vértices) para cada face da célula. Essa é a estrutura de dados que mais ocupa memória num algoritmo de *raycasting*.

VF-Ray melhora o uso da cache, mantendo em memória apenas os dados das faces dos percursos de um conjunto de raios vizinhos. A decisão de quais raios estarão nesse conjunto é feita pela computação da face visível. Os pixels projetados nessa face visível têm alta probabilidade de reutilizarem o mesmo conjunto de faces, para seus percursos ao longo do volume. Esse conjunto de pixels é chamado de *visible set*.

No passo de pré-processamento, os dados do volume são carregados e um conjunto de estruturas de dados é criado na memória. Esse passo é feito apenas uma vez, para todos os diferentes pontos de visualização. As estruturas de dados criadas nesse passo de pré-processamento são:

- $L_v$: uma lista de todos os vértices (pontos);
- $L_c$: uma lista de todas as células, aonde cada célula $c$ tem um ponteiro para as células que compartilham uma face com $c$, descrevendo a conectividade das células;
- $L_{ext}$: uma lista de faces externas do volume.

A renderização de um ponto de observação começa com as operações de rotação dos dados nos eixos $x$, $y$, e $z$, de acordo com o ângulo de visualização. Depois disso, $L_{ext}$ é percorrido para determinar as faces visíveis. As faces visíveis são as faces externas cujas normais fazem ângulo maior que $90^o$ com a direção de observação.

Tendo computado todas as faces visíveis, o Algoritmo 1 apresenta os passos seguintes. Para cada face visível $f_v$, o algoritmo projeta $f_v$ na tela a fim de definir o *visible set* de $f_v$. Para cada raio $r$, que corresponde a um pixel $p$ no *visible set*, o algoritmo tem que computar o ponto de entrada e de saída, $e_{in}$ e $e_{nex}$, do raio em cada célula. A face de saída, $f_{next}$, é computada baseando na face de entrada, $f_{in}$, e na conectividade armazenada em $L_c$. Toda vez que uma nova face é computada, seus coeficientes são guardados em um *buffer* de faces, e a integral de iluminmação de $e_{in}$ para $e_{nex}$ é computada, usando um modelo ótico. Esse passo computa a contribuição da célula na cor e opacidade do pixel $p$.

**Algorithm 1** *loop* principal do VF-Ray

1: Projete $f_v$ na tela
2: **for** cada raio $r$ no *visible set* de $f_v$ **do**
3: $f_{in} \leftarrow f_v$
4: $e_{in} \leftarrow$ interseção do ponto de entrada em $f_v$
5: **repeat**
6: $f_{next} \leftarrow$ FindNextFace ($f_{in}$)
7: $e_{next} \leftarrow$ interseção do ponto em $f_{next}$
8: Armazena os parâmetros de $f_{next}$ no buffer de faces
9: Computa a integral de luminosidade de $e_{in}$ até $e_{next}$
10: $f_{in} \leftarrow f_{next}$ e $e_{in} \leftarrow e_{next}$
11: **until** $r$ sai do volume (dado)
12: Limpa o buffer de faces

Resultados anteriores do VF-Ray [49] mostraram que ele gasta apenas de 1/6 a 1/3 da memória normalmente usada por outras abordagens de *raycasting*, com desempenho comparável.

# Capítulo 4

# O Processador Cell Broadband Engine

Neste capítulo, apresentamos uma descrição detalhada da arquitetura do processador Cell BE e uma descrição sobre a programação para esta nova arquitetura.

## 4.1 A arquitetura do processador Cell BE

O Cell BE é um processador singular. Seu projeto começou em 2000, com a formação da aliança entre IBM, SCEI/Sony e da Toshiba. Em 2001, o STI Data Center foi inaugurado em Austin, Texas e mais de quatro anos depois a sua primeira divulgação foi anunciada. O projeto custou aproximadamente meio bilhão de doláres e contou mais de 500 pessoas. Em 2006, a aliança foi extendida por mais 5 anos.

O Cell BE é uma arquitetura *multicore* heterogênea que combina o tradicional processador PowerPC com múltiplos *mini-cores*, que têm um conjunto de instruções limitado, porém otimizados para instruções SIMD. Um único *chip* contêm um *Power Processing Element* (PPE) e 8 *Synergistic Processing Elements* (SPEs). O nome *synergistic* para esse processador foi escolhido com muito cuidado, já que existe uma dependência

mútua entre o PPE e os SPEs. O SPE depende do PPE para rodar o sistema operacional e, em muitos casos, a *thread* de controle de uma aplicação. O PPE depende dos SPEs para fornecer a maior parte do desempenho computacional de uma aplicação.

A figura 4.1 mostra a arquitetura do processador Cell BE. O PPE, os SPEs, o controlador DRAM, e os controladores I/O estão todos conectados via 4 anéis de dados dentro do *chip*, chamados de *Element Interconnect Bus* (EIB). A seguir, explicaremos o PPE, os SPEs, o EIB e as diferenças entre o PPE e os SPEs com mais detalhes.

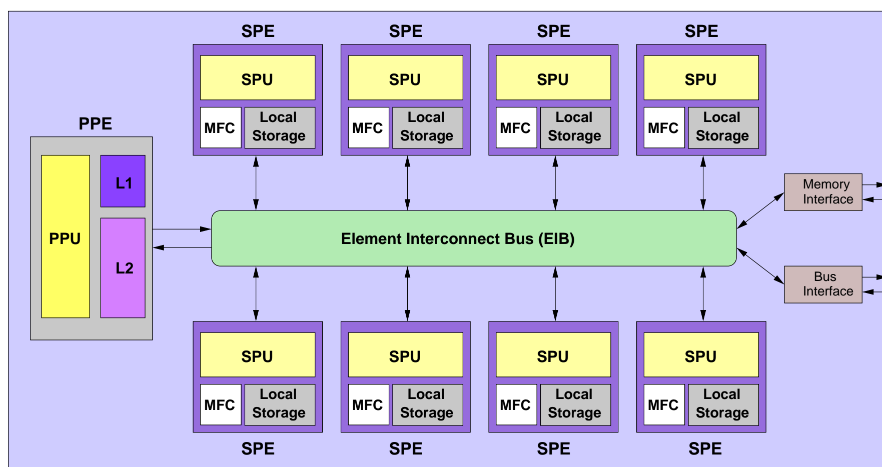


Figura 4.1: Arquitetura do processador Cell BE.

### 4.1.1 O *Power Processing Element* (PPE)

O PPE é um processador PowerPC tradicional, *dual-thread* de 64-bits com uma unidade *Vector Multimedia Extension* (VMX) e dois níveis de *cache*: L1 dentro do *chip*, com 32 KB para instruções e 32 KB para dados, e L2 com 512 KB. O PPE tem um processador PowerPC *Simultaneous multithreading* (SMT) – chamado *PowerPC Processing Unit* (PPU) – que fornece duas unidades de execução independentes para cada camada de *software*. Na prática, os recursos de execução são compartilhados, mas cada *thread* tem

a sua cópia do contexto da arquitetura, como os registradores de propósito geral.

Apesar do *clock* de alta frequencia do PPE, 3.2 GHz, seu propósito é para servir como um controlador e supervisor dos outros núcleos do *chip*.

### 4.1.2 O *Synergistic Processing Element* (SPE)

Os SPEs são os principais motores de computação do processador Cell. Cada SPE é um processador RISC especial com capacidade SIMD de 128-bits que roda um conjunto de instruções específicas para o Cell. Ele é constituído de um núcleo de processamento chamado de *Synergistic Processing Unit* (SPU), um controlador de fluxo de memória (MFC), e uma memória local de 256 KB (*Local Store*). A memória local é usada como área de dados e de código, que será processado pelo SPU, mas não é uma *cache*. Os SPEs não podem acessar a memória principal diretamente, então todo código e dados processado pelo SPE precisa caber na sua memória local. Qualquer dado que o SPE precise, que esteja armazenado na memória principal, precisa ser carregado explicitamente, por *software*, na memória local através de operações DMA (*Direct Memory Access*). Transferência de dados entre a memória local (LS) e a memória principal são regulamentadas pelo MFC através de regras: transferências de dados inferiores a 16 bytes, precisam ser naturalmente alinhadas, ou seja, o endereço de memória precisa ser divisível pelo tamanho da transferência. Além disso, os 4 bits de baixa ordem do endereço do LS precisam ser iguais aos 4 bits de baixa ordem do endereço memória principal, em outras palavras, eles precisam ter o mesmo alinhamento dentro de um quadword. O MFC suporta transferências de 1, 2, 4, 8, 16 e múltiplos de 16 bytes, com máximo de 16 KB. Para transferências iguais ou superiores a 16 bytes, é necessário que os endereços sejam alinhados em 16 bytes. As transferências que não respeitarem essas

regras gerarão uma exceção de *Bus Error*. Uma vez que o dado esteja na LS no SPE, a SPU pode carregá-lo explicitamente nos registradores de propósito geral de 128-bits. O conjunto de instruções da SPU é diferente do conjunto de instruções da PPU e consiste em instruções SIMD de 128-bits. Duas instruções SIMD podem ser executadas por ciclo de *clock* no SPE.

### 4.1.3 O *Element Interconnection Bus* (EIB)

O EIB pode transmitir 96 bytes por ciclo, para uma largura de banda de 204.8 Gigabytes/segundo, permitindo tratar mais de 100 requisições DMA. O EIB é construido por 4 anéis unidirecionais, 2 em cada direção.

### 4.1.4 PPE × SPE

O PPE é rápido na troca de contexto. Os SPEs são mais rápidos em tarefas de computação intensa. Tipicamente, o sistema operacional roda no PPE, enquanto *threads* em user-mode são executadas nos SPEs. Uma diferença significativa entre o PPE e o SPEs está em como eles acessam a memória:

- O PPE acessa diretamente a memória principal, fazendo uso dos *caches* L1 e L2, com instruções de *load* e *store*;
- As instruções de busca dos SPEs e as instruções de *load* e *store* acessam o *Local Store* (LS) de cada núcleo, ao invés da memória principal. Os acessos à memória principal são feitos assincronamente pelo MFC com comandos explícitos de acesso direto a memória (DMA). Essa organização em 3 níveis de armazenamento (registrador, memória local, memória principal), com transferências assíncronas

DMA, ajudam a esconder latências de acesso a memória sobrepondo computação e transferência de dados.

## 4.2 Programando para o processador Cell BE

O processador Cell BE apresenta para o programador um modelo de programação totalmente diferente. O desempenho da aplicação depende do uso efetivo de caracteristicas especiais do processador Cell BE [54]. Enfatizamos aqui os aspectos que distiguem a programação do Cell BE da programação usada em processadores convencionais.

O processador Cell BE pode ser programado usando a linguagem padrão C/C++, utilizando as bibliotecas disponíveis no *IBM Software Development Kit* (SDK) [26]. Uma aplicação de usuário precisa tratar da comunicação, sincronização, e computação SIMD. Um ponto interessante é que uma aplicação já existente rodaria no processador Cell simplesmente recompilando seu código e assim usaria apenas o núcleo do PPE, sem grande esforço, porém sem as vantagens de desempenho que o uso dos SPEs podem oferecer.

Há várias diferenças chaves em programar para o processador Cell comparando-se com a programação *multicore* tradicional para CPUs:

1. O poder dos SPEs vem das operações vetoriais SIMD. Os SPEs não são otimizados para rodar código com dados escalares e manipular dados desalinhados. O de alto desempenho na computação poderá apenas ser atingido se os dados forem organizados de modo que seja apropriado para computações SIMD.

2. Os SPEs não têm memória *cache*. O *engine* DMA é utilizado para acesso explícito a memória através de programação constituindo um trabalho extra quando

comparado à hieraquia de memória normal baseada em *cache*. Todo o código e variáveis precisam ser alocados na memória local. Estruturas de dados grandes da memória principal podem ser acessadas, através de transferências explícitas DMA. Além disso, as chamadas DMA no código precisam ser projetadas para usarem técnicas de *double buffering* ou técnicas similares para evitar bloqueios de processamento esperando pelo término das operações DMA evitando queda no desempenho devido à latência no acesso a dados na memória.

3. Os SPEs não possuem o *hardware* necessário para previsão de desvios. Essa característica dos SPEs permitiu aos engenheiros acoplamento mais *cores* de computação no *chip* do processador, contudo, um desvio custa por volta de 20 ciclos, o que implica que eles devem ser evitados em códigos otimizados para alto desempenho.

Além dessas diferenças, o PPE e os SPEs não têm compatibilidade em seus códigos binários. Dessa forma, podemos resumir que o modelo de programação é um aspecto importante que distingue o processador Cell dos outros. Programar a arquitetura Cell necessita cuidado. No projeto das aplicações, afim de garantir a codificação eficiente, é preciso explorar a capacidade SIMD, garantir que os SPEs estão sendo bem utilizados e que as transferências de dados estejam sendo feitas em paralelo com a computação.

### 4.2.1 Explorando as habilidades SIMD

O processamento de instruções SIMD explora paralelismo no nível de dados, o que significa que operações em todos elementos do vetor são feitos ao mesmo tempo. Isto é, uma única instrução é aplicada para múltiplos elementos de dados em paralelo, como ilustrado na Figura 4.2.

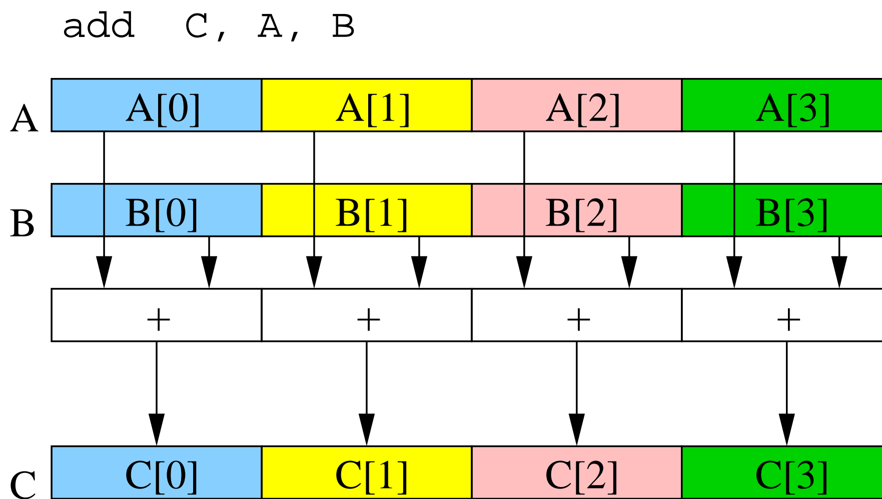


Figura 4.2: Quatro operações de adição em paralelo, em cada elemento dos dois vetores.

Tanto o PPE quanto o SPE suportam operações SIMD. No PPE, elas são providas pelo conjunto de instruções *Vector/SIMD Multimedia Extension* (VMX). No SPE, elas são suportadas pelo SPU *Instruction Set Architecture* (ISA). O processo de preparar um programa para rodar em um processador vetorizado é chamado de vetorização ou "SIMD-ização".

Os SPEs representam a potência computacional por trás do processador Cell BE, e eles são inerentemente processsadores vetoriais. Eles não processam eficientemente operações escalares (não vetoriais).

### 4.2.2 Paralelizando para o SPE

Programas rodando no Cell BE tipicamente dividem o trabalho computacional entre os oito SPEs disponíveis, em cada SPE é designada uma tarefa e dados para serem computados. Do ponto de vista do programador, controlar os SPEs é similar a trabalhar com *threads*. O SDK contêm uma biblioteca, chamada *libspe* (*SPE runtime management library*), que auxilia na construção do código para execução no SPE e na comunicação

entre os *cores* durante a execução. A biblioteca disponibiliza uma interface de programação baixo nível para aplicações, que permite às aplicações acessarem os SPEs e executarem *threads* neles.

Contudo, não é recomendado que uma aplicação faça uso de mais *threads* SPEs do que os SPEs disponíveis. O custo de troca de contexto de um SPE é alto, já que ele necessita guardar maior parte de seus 256 KB de memória local na memória principal, e recarregar o código e os dados da nova *thread*. Essa é a razão pela qual o sistema operacional não é apropriado para rodar no SPE.

Um módulo PPE inicia um módulo SPE criando um contexto para o SPE, usando as chamadas `spe_context_create`, `spe_program_load`, e `spe_context_run` da biblioteca. A função `spe_context_create` cria um contexto para o SPE, que contêm as informações pertinentes sobre uma SPE lógica. Essa informação não deve ser acessada diretamente pela aplicação. Antes de ser habilitado para executar no SPE, o programa precisa ter seu contexto carregado, usando a função `spe_program_load`. O contexto SPE é executado fisicamente no SPE chamando a função `spe_context-_run`. Essa função causa a mudança de contexto da *thread* corrente do PPE para o contexto do SPE, que foi programado para ser executado. Já que o PPE só volta a continuar seu processamento quando os SPEs terminarem suas execuções, *threads* separadas precisam ser criadas no PPE para cada SPE, garantindo assim múltiplas *threads* de execução.

Uma *thread* pode aguardar por eventos (*polling*) ou aguardar o término (*sleep*), esperando pelas *threads* SPE, usando as funções `spe_get_event` ou `spe_wait`. Documentação da biblioteca *SPE Runtime Management* [25, 23] contêm uma descrição detalhada da API disponível para trabalhar com *threads* nos SPEs.

### 4.2.3 Transferências DMA

O processador Cell BE tem uma arquitetura de memória diferente e entender essa arquitetura é a chave para a programação no Cell. O programa SPE referencia sua própria memória (LS) usando o endereço de *Local Store* (LSA - *Local Store Address*). Para o LS de cada SPE é também designado uma extensão de endereços (RA - *Real Address*) da memória principal do sistema. Isso permite que *softwares* com acesso privilegiado mapeiem áreas de LS dentro do espaço de endereços (EA - *Effective address*), onde o PPE, outros SPEs, e outros dispositivos possam acessar esse LS.

O controlador de fluxo de memória, (MFC), é o componente que implementa maior parte da comunicação entre os núcleos do processador Cell BE, incluindo o meio principal de iniciar uma transferência de dados – transferência DMA. Localizado em cada SPE, as interfaces MFCs podem ser acessadas por programas rodando no SPE ou pelo programa rodando no PPE.

O MFC suporta transferência de dados naturalmente alinhados com tamanhos de 1, 2, 4, ou 8 bytes, ou múltiplos de 16 bytes, com tamanho máximo de tamanho de transferência de 16 KB. O desempenho máximo é atingido quando as transferências têm os endereços EA e LSA alinhados em 128-bytes e quando o tamanho da transferência é múltiplo de 128 bytes.

Programas rodando no SPE podem acessar os recursos do MFC através de canais de comunicação (*interface channels*), enquanto que programas rodando no PPE podem acessar os recursos do MFC de um SPE através da interface *Memory-Mapped I/O* (MMIO). As funções do MFC são um conjunto de funções práticas e simples, cada uma executa um comando único DMA. As operações básicas de requisitar e enviar dados de/para a memória principal são `mfc_get` e `mfc_put`, respectivamente. Essas fun-

ções não são bloqueantes, assim o programa irá continuar sua execução depois de emitir esses comandos para o MFC. Essas funções só irão bloquear se a fila de comandos estiver cheia.

Depois que um comando DMA foi iniciado, o programa pode querer esperar que a transação DMA se complete. As três funções principais para isso são: `mfc_write_tag_mask`, `mfc_read_tag_status_any`, e `mfc_read_tag _status_all`. A função `mfc_write_tag_mask` escreve uma máscara de tag que determina de quais tag IDs estamos aguardando a notificação de operação concluída. A função `mfc_read_tag_status_any` espera até qualquer uma das tags especificadas seja concluída, e a função `mfc_read_tag_status_all` espera até que todas as tags especificadas sejam concluídas.

O MFC também suporta um conjunto de comandos de sincronização e de comandos atômicos que podem ser usados para controlar a ordem na qual os acessos ao DMA são feitos. Desses comandos incluem quatro comandos atômicos, três comandos de sinalização (*send-signal*) e três comandos de barreira (*barrier*).

Para mais detalhes, recomendamos a leitura do capítulo *Programming Support for MFC Input and Output* na documentação *C/C++ Language Extensions for Cell BE Architecture* [25, 23].

### 4.2.4 Transferência eficiente DMA e sobreposição de computação

Conforme mencionado anteriormente, as transferências de dados assíncronas com a computação é uma das grandes características da arquitetura Cell BE. Essas transferências assíncronas evitam bloqueios na computação do SPE devido à sobreposição de transferências DMA e computação.

Uma técnica comum de programação paralela que pode ser usada para explorar a sobreposição de transferências de dados com comunicação é a técnica conhecida chamada de *double buffering*, ilustrada na Figura 4.3. Uma transferência sequencial não sobreposta força o SPE a esperar pelos dados antes que ele possa computá-los. A técnica de *double buffering* usa dois *buffers* $B_0$ e $B_1$, no seguinte modo: enquanto o SPE processa o *buffer* de dados corrente em $B_0$, uma transferência assíncrona está em curso, buscando o próximo *buffer* de dados em $B_1$. Se os dois, a transferência e a computação, tomam o mesmo tempo de execução, como mostrado no exemplo da Figura 4.3, todas as transferências (exceto a primeira) ficarão escondidas pela computação e não serão sentidas no tempo final de execução, mostrando que não houve bloqueio na execução no SPE.

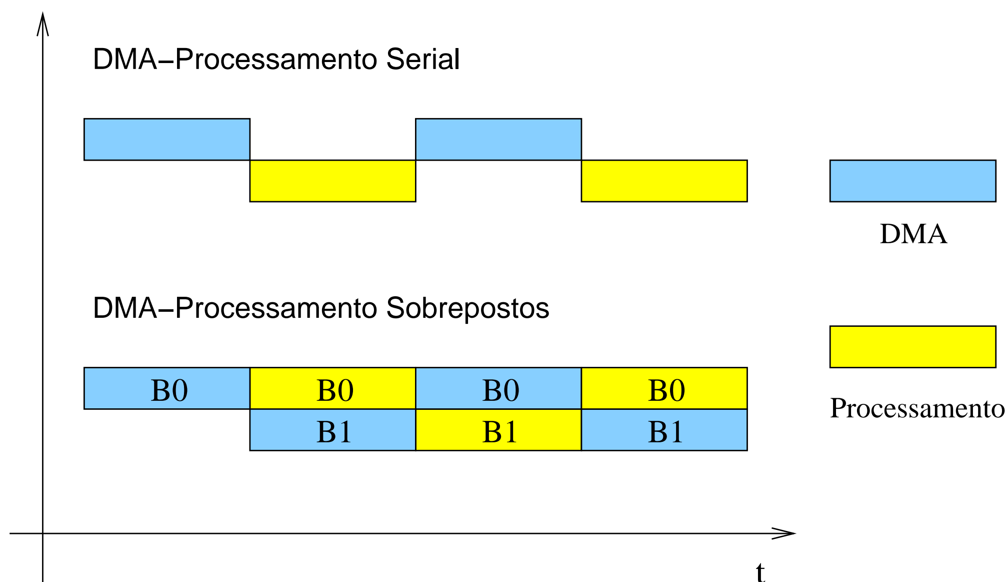


Figura 4.3: Exemplo da técnica de *double buffering*.

### 4.2.5 Programando para o PPE

Aplicações rodando no processador Cell BE normalmente têm pelo menos dois códigos fonte. Um para o PPE e outro para o SPE, já que eles são projetados para rodar diferentes tipos de código, com diferentes conjuntos de instruções. Os dois são capazes de executar instruções SIMD, porém as operações SIMD do PPE são processadas pelo *Vector/SIMD Multimedia Extension unit* (VXU) na PPU, enquanto no SPE, as operações SIMD são executadas pela SPU. Além do conjunto de instruções diferentes, o SPE tem um conjunto de registradores diferente do PPE também, assim, os programas escritos para o PPE e para o SPE usam compiladores diferentes, específicos para cada arquitetura. A Tabela 4.1 mostra um resumo das diferenças entre o PPE e o SPEs.

Tabela 4.1: Diferenças na arquitetura do PPE v SPE [25].

| **Característica** | **PPE** | **SPE** |
|---|---|---|
| **# Registradores SIMD** | 32 (128 bits) | 128 (128 bits) |
| **Registradores** | separados e fixos, float e vector | unificados |
| **Latência de** *load* | variável(*cache*) | fixa |
| **Endereçamento** | $2^{64}$bytes | 256K bytes (local) $2^{64}$bytes (DMA) |
| **Conjunto de instruções** | PowerPC | otimizado para float de precisão simples |
| **Precisão simples** | IEEE 754-1985 | *extended range* |
| *Doubleword* | não há para SIMD | float de precisão dupla - SIMD |

O VXU e seus registradores de 128-bits operam concorrentemente com a unidade de inteiros de ponto fixo (*fixed-point integer unit* - FXU) e com a unidade de execução de ponto flutuante (*floating-point execution unit* - FPU). Assim como as instruções do PowerPC, as instruções VXU são de 4 bytes e alinhados pela palavra, elas têm três ou quatro operandos vetoriais de 128-bits. As instruções VXU suportam execução simul-

tânea em múltiplos elementos que compõem um operando vetorial de 128-bits. Esses elementos do vetor podem ser byte, halfword, ou word.

As instruções foram escolhidas para serem úteis em algoritmos de processamento de sinais digitais, incluindo computação gráfica 3D. As instruções são divididas nos seguintes tipos:

- *Vector Integer Instructions* – Incluem instruções vetoriais de aritmética, de comparação, de operações lógica, de rotação e deslocamento. Elas operam nos elementos vetoriais do tipo: byte, halfword, e word.

- *Vector Floating-Point Instructions* – Incluem instruções de aritmética de ponto flutuante, multiplica/addiciona, arredondamento e conversão, comparação, e de estimativas. Elas operam em elementos de vetores do tipo ponto flutuante de precisão simples (SPFP).

- *Vector Load and Store Instructions* – Incluem instruções apenas para carregar e restaurar dados inteiros ou de ponto flutuante em vetores. Não há instruções de atualização ou alteração. Elas operam em vetores de 128-bits.

- *Vector Permutation and Formatting Instructions* – Incluem instruções de empacotamento, *merge*, *splat*, *permute*, *select*, e *shift*.

- *Processor Control Instructions* – Incluem instruções que leem e escrevem o registrador de status e controle vetorial (VSCR).

- *Memory Control Instructions* – Instruções para controle de *caches* (*user-level* e *supervisor-level*). Essas instruções são "no-ops"(*No Operation Performed*).

Um conjunto de extensões (*intrinsics*) da linguagem C está disponível para programação vetorial SIMD para PPE [28]. Essas extensões incluem novos tipos de dados

vetoriais e um grande conjunto de funções (*intrinsics*) escalares e vetoriais. As *intrinsics* VMX usam o prefixo `vec_`. Na Listagem 4.1, mostramos um exemplo de um programa bem simples que ilustra como é fácil o uso de instruções vetoriais dentro de um programa PPE. A instrução `vec_add` soma os vetores $v1$ e $v2$.

```
#include <stdio.h>
typedef union {
  int iVals[4];
  vector signed int myVec;
} vecVar;
int main() {
  vecVar v1, v2, v3; // variaveis
  // carregamos valores para os vetores v1 e v2
  v1.myVec = (vector signed int){2, 2, 2, 2};
  v2.myVec = (vector signed int){10, 20, 30, 40};
  // somamos os vetores usando a funcao intrinsic vec_add
  v3.myVec = vec_add( v1.myVec, v2.myVec );
  printf( ``Soma: %d, %d, %d, %d'',
          v3.iVals[0], v3.iVals[1],
          v3.iVals[2], v3.iVals[3] );
  return 0;
}
// Resultado de saida:
// Soma: 12, 22, 32, 42
```

Listing 4.1: Exemplo do uso funções *intrinsics* do VXU do PPE.

### 4.2.6 Programando para o SPE

O SPE suporta os dois tipos de operações em ponto flutuante, de precisão simples e dupla. Instruções de precisão simples são executadas em vetores que suportam 4 elementos, fazendo uso total do *pipeline*, enquanto as instruções de precisão dupla fazem uso parcial do *pipeline*, e um vetor comporta 2 elementos. O formato dos dados para instruções de precisão simples e dupla são definidos pelo padrão *IEEE Standard 754*, mas os resultados calculados pelas instruções de precisão simples não são totalmente compatíveis com o padrão.

O Local Store (LS) do SPE pode ser considerado como um *cache* controlado por *software* que pode ser preenchido e esvaziado através de comandos de transferência de dados DMA. Ele suporta tanto instruções (código) quanto dados. Quando existe concorrencia de acesso ao LS, a SPU arbitra o acesso de acordo com as seguintes prioridades: primeiro, comandos DMA de leitura e escrita emitidos pelo PPE ou por um dispositivo de I/O; segundo, *loads* e *stores* da SPU; e por fim, *prefetch* de instruções.

A SPU tem dois *pipelines*, chamados de *even* (*pipeline* 0) e *odd* (*pipeline* 1). Nesses *pipelines*, a SPU pode emitir e completar até duas instruções por ciclo, uma em cada *pipeline*. Emissão dupla ocorre quando um grupo de "busca-instrução"tem a possibilidade de emitr instruções que suportam dupla emissão, assim a primeira instrução pode ser executada no *pipeline even* e a segunda pode ser executada no *pipeline odd*.

Para a comunicação com o PPE, a arquitetura oferece três mecanismos principais:

- **DMA:** Para transferir dados entre a memória principal e o LS.
- *Mailbox***:** Para controlar a comunicação entre o SPE, o PPE e outros dispositivos. *Mailboxes* suportam mensagens de 32-bits. Cada SPE tem dois *mailboxes* para mensagens de envio e um mailbox para recebimento de mensagens.

- *Signal Notification*: Para controlar comunicação vinda do PPE ou de outros dispositivos. *Signal notification* (também conhecido como sinalização) usa registradores de 32-bits que podem ser configurados para sinalização *one-sender-to-one-receiver* ou sinalização *many-senders-to-one-receiver*.

Todas três comunicam-se através de registradores MMIO mantidas pelo *Memory Flow Controller* (MFC) em cada SPE.

Existem 204 instruções no *Instruction Set Architecture* (ISA) da SPU, e elas são agrupadas em 11 classes de acordo com suas funcionalidades [27]. As classes são: *load* e *store* de memória, formação de constantes, operações lógicas e inteiras, *shift* e *rotate*, comparações, *branch* e *halt*, *hint-for-branch*, *floating point*, controle, SPU *channel*, interrupções SPU, sincronização e ordenação.

Como o SPE é especializado em operações de vetores de 128-bits, é necessário e útil agrupar os dados em vetores SIMD. Em aplicações 3D, por exemplo, cada vértice ($v_0$, $v_1$, $v_2$) de um triângulo pode ser guardado como coordenadas homogêneas ($x$, $y$, $z$, $w$) em um vetor SIMD. Nesse caso, como o quarto elemento de cada vetor não é usado e para cada triângulo, há desperdício de memória de 1 vértice, ou 1 elemento do vetor.

Para operações escalares, o SPE usa um esquema chamado *preferred slot* nos registradores vetoriais de 128-bits, como mostrado nas áreas escuras da Figura 4.4. Para usar o esquema de *preferred slot* necessitamos operações extras para deslocar o elemento para o *preferred slot* e então deslocar de volta para a localização original, o que explica por que os SPEs não são apropriados para operações escalares.

Um conjunto de extensões (*intrinsics*) da linguagem C também está disponível para programação SPE. As extensões fornecem para os programadores controle explícito das instruções SIMD sem necessidade de acesso direto aos registradores ou programação

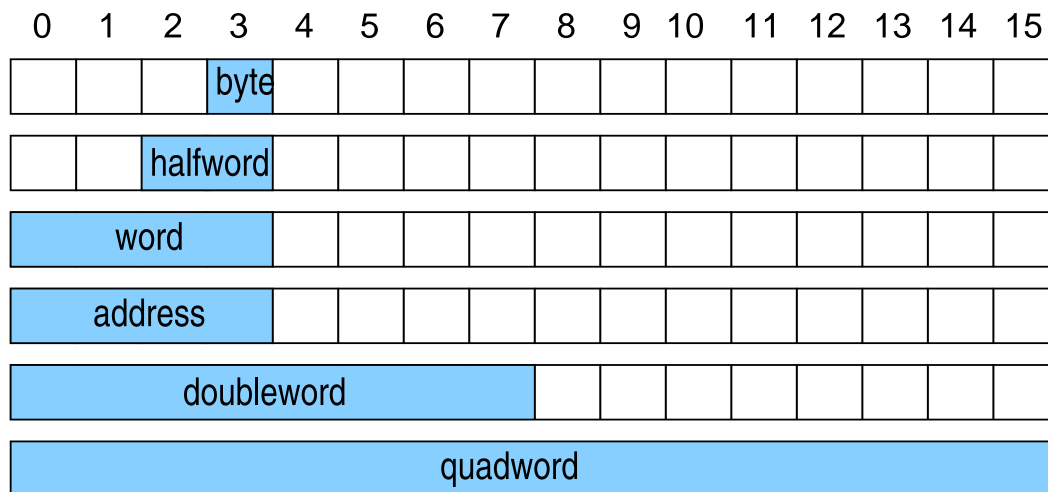


Figura 4.4: Esquema dos tipos de dados e dos *slots* preferidos.

*assembly*. As intrinsics na SPU usam o prefixo `spu_`. Por exemplo, o SDK e API da SPU fornecem a função *intrinsic* `t = spu_add(a, b)` que substitui a instrução *assembly* `fa rt, ra, rb`. Na Listagem 4.2, mostramos o mesmo programa exemplo ilustrado na programação para PPE, contudo o código foi alterado para rodar no SPE.

```
#include <stdio.h>
typedef union {
  int iVals[4];
  vector signed int myVec;
} vecVar;
int main() {
  vecVar v1, v2, v3; // variaveis
  // // carregamos valores para os vetores v1 e v2
  v1.myVec = (vector signed int){2, 4, 6, 8};
  v2.myVec = (vector signed int){1, 2, 3, 4};
  // somamos os vetores usando a funcao intrinsic spu_add
  v3.myVec = spu_add( v1.myVec, v2.myVec );
  printf( ``Soma: %d, %d, %d, %d'',
```

```
          v3.iVals[0], v3.iVals[1],
          v3.iVals[2], v3.iVals[3] );
  return 0;
}
// Resultado de saida:
// Soma: 3, 6, 9, 12
```

Listing 4.2: Exemplo do uso funções *intrinsics* do SPE.

### 4.2.7 Orientações básicas no desenvolvimento de código

Resumimos aqui várias orientações básicas para desenvolvimento de código para o processador Cell BE:

- Execute o código principal no PPE;
- Paralelize o código para os SPEs – mantendo o acesso a memória sequencial e não vetorize ainda;
- Vectorize o código no SPE;
- Prepare seus dados para processamento no SPE – alinhamento dos dados;
- Inclua técnicas de sobreposição nas transferências DMA;
- Evite desvios nos caminhos críticos de execução.

### 4.2.8 Dicas de implementação

Além das importantes práticas de programação apresentadas até agora, existem algumas dicas extras de programação apresentadas por Brokenshire [6], que podem ser úteis

para programadores, a fim de melhorar o desempenho de suas aplicações baseadas no processador Cell:

1. **Descarregar o máximo de trabalho possível nos SPEs** – SPEs representam o poder de computação do Cell BE, então use o PPE como um processador de controle, para comandar a execução dos SPEs.

2. **Escolha uma estratégia de particionamento e alocação de trabalho que minimize o uso de operações atômicas e eventos de sincronização** – Essa é uma prática comum em programação parelela, para o Cell BE uma possível estratégia é os SPEs particionarem o trabalho algoritmicamente. Por exemplo, considere uma aplicação de processamento de imagem, na qual a varredura das linhas é feita por $n$ SPEs. Cada SPE pode algoritmicamente computar sua parcela de trabalho dividindo o número de linhas para serem computadas por $n$.

3. **Acomodar as diferenças potenciais nos tipos de dados** – O PPE e os SPEs podem ter tipos de dados com diferentes tamanhos. O PPE pode tanto ser ILP32 ou LP64 (inteiros são 32 bits, *longs* e ponteiros são 64 bits), enquanto os SPEs são sempre ILP32. Assim, aplicações de 64-bits devem ter cuidado quando tratarem estruturas de dados compartilhadas entre PPE e os SPEs.

4. **Explorar** *multithreading* **no PPE** – O PPE é *multithread*, que pode ter 2 *threads* aonde são mantidos o estado completo da arquitetura do processador. Então, é altamente encorajador o uso de *multithread* no PPE quando estiverem acontecendo significantes *cache misses* L1 e L2 nas *threads*, ou quando o código das *threads* tiver dependência de operações de ponto flutuante.

5. **Projetar estruturas de dados para acesso eficiente** – Para conseguir acessos eficientes dos SPEs, programadores devem considerar alinhamento de dados, padrões de acesso e de localização nas estruturas de dados.

6. **Inicie as operações DMA pelo SPE** – Ao invés do PPE empurrar os dados para o SPE, deixe que o SPE puxe os dados usando o MFC para iniciar as operações DMA. Isso deve ser feito para evitar engargalamentos no PPE, e por que o número de ciclos para iniciar uma transferência do SPE é menor do que o número de ciclos para iniciar a mesma transferência do PPE.

7. **Mantenha-se no** *chip* – Se a sua aplicação permitir, tente organizá-la de modo que use o máximo possível o *Local Store* do SPE, e compartilhe os dados entre os SPEs, evitando comunicação com a memória principal.

8. **Evite operações escalares no SPE** – Lidar com operações escalares necessita várias operações dependentes, uma vez que os dados são geralmente transportados para elementos de vetores (*preferred slots*) para computação.

9. *Unroll* **e** *pipeline* **para** *loops* – Deve ser removido todos os *loops* em que o número de iterações é conhecido e constante, sequencializando e reescrevendo o código. Tendo um grande numéro de registradores, os SPEs são hábeis em fazer *unroll* de *loops* de tamanho considerável.

10. **Evite multiplicações de inteiros** – Deve ser evitado a execução de multiplicações de inteiros de 32-bits, já que o SPE contêm apenas multiplicador de 16x16 bits.

11. **Considere computar ao invés de resultados pre-computados** – Uma estratégia comum que muitos programadores usam para aumentar o desempenho de uma

aplicação, chamada de tabela de valores pré-computados, não é eficiente em programas para o SPE, já que essas tabelas e valores não são apropriadas para SIMD e consomem espaço do LS. Se possível, é melhor que se compute esses valores novamente, ao invés de lê-los da memória.

12. **Projete para uma memória local limitada** – O LS do SPE é limitado em 256KB, distribuido para instruções do programa, *stack*, estruturas de dados locais e *buffers* do DMA.

## 4.3 Ambiente de desenvolvimento

Para permitir a um programador tirar vantagem da arquitetura do processador Cell BE, a IBM desenvolveu um Kit de Desenvolvimento de *Software*, *Software Development Kit* (SDK) [26], que fornece bibliotecas, ferramentas, e recursos para o desenvolvimento e ajustas aplicações para a tecnologia. Não apenas incluem ferramentas para desenvolvimento, mas também um ambiente de simulação capaz de rodar um *kernel* do Linux com as extensões da arquitetura Cell BE. Entre os muitos componentes disponíveis no SDK [25, 23], destacamos:

- Ferramentas GNU incluindo compiladores C/C++ (*gcc*), *linkers*, *debuggers* (*gdb*), *assemblers* e utilitários binários para PPU e SPU;

- Compiladores IBM, *xlc* para C/C++ e *xfl* para Fortran, para PPU e SPU;

- Bibliotecas padronizadas de operações matemáticas SIMD para os vetores do PPU e do SPU;

- Um conjunto de ferramentas de desempenho, como *oprofile*, *CellPerfCount*, *FDPR-Pro*, *CodeAnalyzer* e *spu_timing*;
- *SPE Runtime Management Library*, biblioteca que fornece interfaces de programação padronizadas de baixo nível para acesso especial aos SPEs;
- Um ambiente de desenvolvimento integrado (IDE - *Integrated Development Environment*) para Eclipse;
- O IBM *Full-System Simulator*, aplicação que emula o comportamento de um sistema completo, baseado no sistema operacional Linux, que contenha um processador Cell BE;
- Código fonte contendo exemplos de programas, bibliotecas, *benchmarks* e aplicações demonstrativas.

## 4.4 SystemSim, O Simulador IBM do Cell BE

O *IBM Full-System Simulator* (SystemSim) é o simulador desenvolvido pela IBM que fornece simulação completa de sistemas com processadores baseados na arquitetura IBM PowerPC, incluindo o Cell BE. Também chamado de "Mambo", contêm inúmeras características para auxiliar no projeto e análise de *software*. O SystemSim é amplamente usado dentro da IBM como ferramenta de suporte no desenvolvimento de sistemas operacionais, compiladores, componentes de *software* para novos sistemas bem antes da disponibilidade do *hardware*.

O simulador oferece diferentes modos de simulação, variando de simulações funcionais do processador para simulações de análise de desempenho de aplicações, os modos disponíveis são:

- *Simple* (*functional-only*) – modela o efeito das instruções, sem se preocupar com a exatidão do tempo necessário para executá-las. Nesse modo, uma latência fixa é atribuida para cada instrução, e ela pode ser alterada pelo usuário do simulador. Já que a latência é fixa, efeitos causadores de latência não são levados em conta. O modo *functional-only* assume que o acesso a memória é síncrono e instantâneo. Esse modelo é útil no desenvolvimento de *software*, quando uma medida de tempo de execução não é necessária.

- *Fast* – é similar com o modo *simple* (*functional-only*), já que ele também modela os efeitos das instruções enquanto não se atenta a precisão do modelo de tempo de cada instrução executada. Porém, o modelo *fast* desconsidera análises padrões fornecidas pelo modo *Simple*, como análises estatísticas, *triggers*, geração de relatórios. Modo *fast* é pretendido para ser usado numa simulação rápida sobre partes desinteressantes de análise mais profunda.

- *Cycle* (*performance*) – esse modo é o mais completo, que não só é fiel a funcionalidade mas como também ao tempo de execução de cada instrução. Ele considera políticas de execução interna (prioridades) e de temporização, assim como os componentes do sistema, como filas e *pipelines*. Operações levam vários ciclos para serem concluídas, já que são considerados o tempo de processamento e restrições de recursos do sistema.

# Capítulo 5

# Implementando o VF-Ray no Cell BE

Neste capítulo, apresentamos a descrição de como o algoritmo VF-Ray foi implementado explorando as características do processador Cell BE. Descrevemos os desafios encontrados durante a programação da aplicação, como exploramos as facilidades SIMD da arquitetura, como distribuímos o trabalho entre as SPEs, como orquestramos as transferências de dados pelo DMA e alguns outros detalhes de nossa implementação.

## 5.1 Desafios

Os desafios principais na implementação do VF-Ray no processador Cell BE são:

- estruturas de dados precisam ser manualmente alinhadas para segmentos de 128 bytes;
- as computações mais pesadas (que consomem mais tempo) precisam usar recursos SIMD;
- estruturas de dados que guardam as informações usadas na renderização não ca-

bem no LS dos SPEs, então transferências DMA da próxima célula que será atravessada pelo raio precisam ser orquestradas de modo a evitar bloqueios de processamento nos SPEs;

- todos os desvios críticos *if-then-else e controles de loop* que executem nos SPEs precisam ser eliminados, já que os desvios provocam uma queda elevada no desempenho dos SPEs.

O VF-Ray é implementado como uma aplicação C++, com mais de 20 classes e mais de 10.000 linhas de código. Dada sua complexidade, a implementação no Cell foi feita numa abordagem passa-a-passo, reusando o código original o máximo possível.

O primeiro passo foi ajustar as estruturas de dados. Todas as estruturas de dados foram alinhadas na memória. Estruturas com número arbitrário de elementos foram transformadas em vetores de 128-bits. Por exemplo, a `class Point` que compreende 4 elementos do tipo float (as coordenadas de cada ponto no *grid* seguido de seu valor escalar, $\alpha$) foi transformada num `vector float` de 128-bits. Do mesmo modo, a `class Cell` com 4 inteiros sendo os índices de seus vértices e outros quatro inteiros sendo índices de suas quatro células vizinhas, foram substituídos por dois `vector signed int` de 128-bits.

Depois disso, efeatuamos os seguintes passos: exploramos as facilidades SIMD, parelelizamos a computação do raio nos SPEs, e orquestramos as transferências DMA. Cada um desses passos serão explicados em detalhes a seguir para duas abordagens adotadas. A primeira, seguimos fielmente as recomendações da documentação e computamos exaustivamente todos os dados, sem armazernar nada na memória, essa abordagem foi chamada de Cell-VF-Ray. Na segunda abordagem, usamos um pequeno *buffer* nos SPEs para guardar faces que já foram computadas, essa abordagem foi chamada de

Cell-VF-Ray-FB.

## 5.2 O Algoritmo Cell-VF-Ray

No algoritmo Cell-VF-Ray, seguimos as recomendações de evitar desvios e estruturas de dados grandes nos SPEs, explorar transferências de dados assíncronas, e usar instruções SIMD para computação de dados em paralelo. Dessa forma, o algoritmo foi projetado transferindo a responsabilidade de computar cada pixel, isto é, cada raio, para os SPEs. A PPE organiza e distribui a computação, os SPEs são notificados de que existem dados para serem computados e passam a requisitá-los e processá-los, no final do processamento escrevem o resultado na memória principal e esperam pelo próximo dado.

Dessa forma, a memória do SPE se armazena apenas os *buffers* usados nas computações, não armazenando nada sobre computações anteriores. Usamos operações SIMD nas computações, como é melhor detalhado nas seções seguintes. Todas as transferências de dados entre os SPEs e o PPE ocorrem em paralelo com a computação. Evitamos o uso de desvios ao máximo no código do SPE.

A seguir descrevemos com detalhes as técnicas usadas no algoritmo.

### 5.2.1 Paralelizando para o SPE

A parte do VF-Ray que mais consome processamento está entre as linhas 5–11 do Algoritmo 1. Como a computação de cada raio é independente no processo de *raycasting*, distribuímos a computação dos raios entre os SPEs. Contudo, as estruturas de dados $L_v$, e $L_c$, que descrevem a conectividade das células, não cabem no LS do SPE. Então, propomos aqui um esquema de *pipeline* para alimentar os SPEs com os dados apropriados

para evitar bloqueios.

Inicialmente, o PPE determina as faces visíveis para o ponto de visualização atual. O PPE projeta cada face visível na tela, criando seu *visible set*. Os pixels dentro de cada *visible set* são, então, distribuídos entre os SPEs para a computação do raio. Depois disso, o PPE é responsável por orquestrar o envio dos raios para os SPEs. O Algoritmo 2 mostra o código executado no PPE.

**Algorithm 2** Algoritmo do PPE

1: Rotaciona os dados para o ponto de visualização atual
2: Procura pelas faces visíveis
3: **for** cada face visível $f_v$ **do**
4: Projeta $f_v$ na tela e determina seu *visible set*
5: **for** cada ray $r$ no *visible set* de $f_v$ **do**
6: Envia $r$ para uma SPE desocupada

O SPE é responsável pela a computação da entrada e da saída do raio em cada célula, assim como a integral de iluminação. Contudo, o SPE não tem na memória a informação sobre todas as células que o raio irá interceptar. Para uma célula, o SPE pode computar os pontos de entrada e saída do raio, $e_{in}$ e $e_{next}$. A face que contem o ponto de saída indica a próxima célula que será atravessada. Então, a computação do SPE segue o *pipeline*. O SPE requisita a informação sobre a próxima célula que será atravessada, e enquanto o DMA está transferindo os dados para o LS, o SPE pode computar a integral de iluminação para o par $e_{in}$ e $e_{next}$. Esse processo é repetido, até que o raio deixe o volume. Nesse caso, o SPE irá requerer outro pixel para o PPE, e toda computação é repetida. O Algoritmo 3 detalha o código executado no SPE.

**Algorithm 3** Algoritmo do SPE

1: **while** existe um raio $r$ enviado pelo PPE **do**
2: Encontre a face de entrada $f_{in}$ do raio $r$
3: Encontre o ponto de entrada $e_{in}$ da face $f_{in}$
4: **repeat**
5: $f_{next} \leftarrow$ FindNextFace ($f_{in}$)
6: $e_{next} \leftarrow$ ponto de interseção na face $f_{next}$
7: Requer informações sobre a próxima célula
8: Compute integral de iluminação de $e_{in}$ até $e_{next}$
9: $f_{in} \leftarrow f_{next}$ e $e_{in} \leftarrow e_{next}$
10: **until** $r$ sair do volume (dados)

### 5.2.2 Explorando as facilidades SIMD

No processo de renderização existem alguns procedimentos que podem ser altamente otimizados com uso das instruções vetoriais disponíveis no processador Cell BE. O procedimento que normaliza e centraliza os dados de acordo com as dimensões da tela é um deles. Esse procedimento realiza a translação seguido de uma multiplicação de todos os pontos por um fator de normalização. Esse fator é o inverso da diagonal do *bounding box*, circundando os dados (o volume) nesse espaço. Tanto a translação quanto a multiplicação são operações que podemos combinar numa única instrução vetorial `vec_madd`, que se realizará sobre os 4 elementos do vetor em paralelo.

Na função *FindNextFace*, temos que computar os coeficientes de todas as três faces restantes da célula, já que desejamos achar a face de saída do raio, $f_{next}$. Esse processo foi vetorizado computando os coeficientes das três faces de uma vez.

Na função de integral de iluminação, exploramos o fato que a mesma operação é feita nos três componentes de cor do pixel, $R$, $G$, e $B$. Então, esses três componentes foram empacotados em um vetor, de forma que pudéssemos computá-los uma única vez para os 3 componentes.

### 5.2.3 Transferências DMA

A paralelização proposta na seção 5.2.1 irá apenas funcionar corretamente se os SPEs puderem ser preenchidos constantemente com dados sobre a próxima célula. Resolvemos esse problema criando uma *pipeline*, onde os SPEs mantêm a computação de um par de interseções, enquanto os dados da próxima célula estão sendo transferidos pela DMA. Então, as linhas 7 e 8 no Algoritmo 3 ocorrem em paralelo, sobrepondo a computação e a comunicação. Esse *pipeline* funciona bem por que cada transferência DMA move dados entre a memória principal e o LS de cada SPE numa maneira assíncrona. Além do mais, o DMA permite a requisição de múltiplos blocos de memória em uma operação. Cada SPE têm 32 canais de comunicação que controlam sincronização e sinalização com o PPE. As operações básicas utilizadas na requisição e de envio de dados de/para a memória principal foram: `mfc_get` e `mfc_put`, respectivamente.

### 5.2.4 Evitando Desvios

Como a função *FindNextFace* é a que mais consome tempo de computação no código do SPE, fomos cuidadosos nos detalhes da sua implementação. Além de vetorizar o código, também removemos desvios *if-then-else* da computação da função, afinal desvios têm um custo computacional alto nos SPEs.

O segredo para eliminar desvios é explorar as instruções de seleção de bits. Isto é, um comando *if-then-else* pode ser eliminado processando os dois casos e selecionando o resultado, como se fosse uma função condicional.

A Listagem 5.1 mostra um exemplo de remoção de desvio na SPU. A primeira parte do código usa o método normal de *if-then-else*, enquanto a segunda parte evita o desvio. Todas as variáveis são vetoriais. O comando `spu_cmpgt` atribui a variável `test` para

1 se `value < 0`; e para 0, se não. O comando `spu_sel` selecionará o primeiro ou o segundo parâmetro, dependendo do valor de test, atribuindo o resultado da seleção para a variável vetorial do lado esquerdo do sinal de igual.

```
// Codigo normal com branching
if (value < 0)
  temp = p0;
  p0 = p1;
  p1 = temp;

// Evitando branching
test = !spu_cmpgt(value, zero);
temp = p0;
p0 = spu_sel(temp, p1, test);
p1 = spu_sel(temp, p1, spu_andc(all,test));
```

Listing 5.1: Eliminando *branching* no SPU.

Outra técnica usada para evitar desvios foi o *unroll* de *loops*, nela abrimos os laços de repetição que têm um número finito e conhecido de repetições, dessa forma aproveitamos melhor o grande número de registradores disponíveis na arquitetura do SPU, e não precisamos ficar testando a condição de fim do *loop*. Ao invés de usarmos o *loop*, substituimos por código sequencial, deixando assim, o código maior em número de linhas, no entanto, mais eficiente para ser executado no SPE.

A Listagem 5.2 mostra uma função escrita de forma convencional e evitando *loops*.

```
// Funcao normal
void GetFaceVerticesIndices( PCELL c, unsigned int nFaceIdx, /* [OUT]
    */unsigned int *nVertices )
{
    unsigned int i;
```

```
    nVertices[0] = TETRAfaces[ nFaceIdx ][ 0 ];

    for( i = 1 ; i < nVertices[0]+1 ; i++ )
    {
        nVertices[i] = c->ptIdx[ TETRAfaces[ nFaceIdx ][ i ] ];
    }
}

// Unroll de loops
void GetFaceVerticesIndices( PCELL c, unsigned int nFaceIdx, /* [OUT]
     */unsigned int *nVertices )
{
    nVertices[0] = TETRAfaces[ nFaceIdx ][ 0 ];
    /* loop unroll */
    nVertices[1] = c->ptIdx[ TETRAfaces[ nFaceIdx ][ 1 ] ];
    nVertices[2] = c->ptIdx[ TETRAfaces[ nFaceIdx ][ 2 ] ];
    nVertices[3] = c->ptIdx[ TETRAfaces[ nFaceIdx ][ 3 ] ];
}
```

Listing 5.2: *Unroll* de *loops* no SPU.

## 5.3 Algoritmo Cell-VF-Ray-FB

O algoritmo Cell-VF-Ray se baseou no princípio de não aproveitar computações anteriores, devido à capacidade limitada de memória do SPE. A idéia é recomputar quando necessário, conforme recomendado pela documentação do Cell BE.

Embora tenhamos obtido ganhos consideráveis de desempenho com essa estratégia,

ocupamos menos de 50% do espaço disponível do LS do SPE. Como o índice de reaproveitamento de faces é enorme, e os ganhos, já obtidos em trabalhos anteriores, por esse reaproveitamento é considerável, decidimos implementar uma nova versão do algoritmo, que tira proveito do reaproveitamento de faces. Essa nova versão é chamada de Cell-VF-Ray-FB.

A idéia básica do algoritmo Cell-VF-Ray-FB é aproveitar resultados da computação de faces, já que raios vizinhos geralmente utilizam-se de faces em comum na sua computação, criamos um *buffer* de faces e seus coeficientes já computados. Como também não podemos gastar tempo fazendo procura nesse *buffer*, precisamos de um método direto de achar se a face já foi computada e quais são seus valores. Para conseguir a localização sem necessidade de realizar uma procura, que teria custo computacional, utilizamos os valores dos vértices da célula como referência na fórmula apresentada na Listagem 5.3. A cada mudança de face visível no PPE, limpamos esse *buffer* de faces nos SPEs.

```
hashIdx  = (cell.ptIdx[0] << 24 & 0xFF000000) |
      (cell.ptIdx[1] << 16 & 0x00FF0000) |
      (cell.ptIdx[2] <<  8 & 0x0000FF00) |
      (cell.ptIdx[3]       & 0x000000FF);
hashIdx \%= FACESBUFFERSIZE;
```

Listing 5.3: Index do FacesBuffer

O algoritmo Cell-VF-Ray-FB é similar ao algoritmo Cell-VF-Ray. No PPE nada foi alterado, contudo, no SPE a função *FindNextFace* do Algoritmo 3 foi alterada da seguinte forma: antes de computar as faces da célula, ela procura no FacesBuffer local do SPE para saber se as faces já foram computadas. Caso já tenham sido computadas, nenhum novo cálculo é realizado. Caso contrário, as faces são computadas e armazenadas

no FacesBuffer.

Em termos de estratégia de alto desempenho utilizadas, como *pipeline* na transferência das células para o LS, a redução dos desvios na função *FindNextFace* e o *unroll* de *loops*, o algoritmo Cell-VF-Ray-FB utiliza as mesmas estratégias do algoritmo Cell-VF-Ray.

# Capítulo 6

# Resultados Experimentais

Neste capítulo, apresentamos nossos resultados experimentais. Primeiramente, apresentamos o ambiente de teste utilizado e, em seguida, os resultados obtidos na execução no PS3. Os primeiros resultados obtidos mostraram o potencial do processador Cell para acelerar o *raycasting*. Entretanto, continuamos a busca para melhorar o desempenho de nossa implementação e utilizamos um simulador do processador Cell para fazer uma análise detalhada do desempenho. A partir desta análise, pudemos propor novas melhorias no algoritmo. Estas melhorias e seus resultados são apresentados na última seção.

## 6.1 Ambientes de Teste

Nossos experimentos foram realizados em dois ambientes de testes diferentes: Sony Playstation 3 (PS3) e o Simulador Cell – SystemSim.

O console de video-game Playstation 3 permite que executemos 2 sistemas operacionais, um chamado GameOS e outro OtherOS, aonde podemos instalar Linux. Em

nossos testes usamos Fedora Core 9, como OtherOS. O PS3 contém um processador Cell, 3.2 GHz, e 256 MB de RAM. No entanto, o PS3 opera com apenas 6 das 8 SPEs do processador. Nossa abordagem do VF-Ray para Cell foi escrita em C++ usando as extensões SIMD incluídas no Kit de Desenvolvimento de *Software* [26].

Já o simulador, SystemSim, foi instalando numa máquina com arquitetura Intel e sistema operacional Fedora Core 9, com 3.0 GHz e 2 GB de RAM. Ele é totalmente parametrizável, permitindo a escolha do número de SPEs disponíveis, modo de operação de execução de código, entre outros.

### 6.1.1 Configuração do Ambiente

O PS3 é mostrado na Figura 6.1. Para compilar e executar códigos compatíveis com a arquitetura Cell BE no PS3, tivemos que configurar um ambiente de programação no console de video-game. O primeiro passo é instalar o *OtherOS bootloader*, que permite a instalação do sistema operacional Linux no PS3. Atualmente, o IBM SDK [26] pode apenas ser usado em um Linux OS. Para o *bootloader* e o *kernel* do Linux funcionarem adequadamente, o *Firmware* do PS3 precisa estar atualizado, o que pode ser feito no menu principal do PS3, selecionando a opção *System Update*. Mais informações sobre a configuração de sistema pode ser encontrada no Manual do Usuário do PS3 [46].

A instalação do sistema operacional Linux no PS3 varia dependendo da distribuição Linux usada, por essa razão restringimos essa descrição para distribuição usada no nosso trabalho, Fedora Core [24]. Fedora suporta o hardware do PS3 desde a versão Fedora 5, enquanto o IBM SDK foi disponibilizado para a distribuição Fedora apartir do *release* 7.

Para realizar o primeiro passo, precisamos de uma mídia de armazenamento, como

um USB *flash drive* (*pen-drive*) com *file system* FAT ou um CD/DVD. A mídia é usada para armazenar o *bootloader* no caminho específico (*PS3/otheros/otheros.bld*). O segundo passo é instalar o *bootloader*: ligue o PS3 com a mídia contendo o *bootloader* inserida; selecione *Settings / System Settings / Install Other OS* no menu principal do PS3; prossiga com a instalação do *bootloader*. Depois de instalado, selecione *Other OS* como o sistema padrão *Default System* no menu *System Settings*. O sistema irá começar com o *bootloader* instalado no próximo *reboot*.

O passo final de instalação é inserir o disco do Fedora Core no PS3, e instalá-lo. Uma vez a instalação é terminada, o PS3 já está pronto para ser usado como um PC normal, com um processador Cell BE ao invés de uma CPU convencional. A única configuração que falta é a instalação do IBM SDK. Até o presente momento, o IBM SDK suporta o pacote de desenvolvimento do Fedora Core 9, que usamos com sucesso em nossos experimentos.

Figura 6.1: Playstation 3: Arquitetura Cell BE.

## 6.2 Massas de Dados

Para avaliar o desempenho dos algoritmos Cell-VF-Ray e Cell-VF-Ray-FB usamos sete dados tetraedrais que são amplamente usados na literatura: SPX, Blunt Fin, Oxygen

Post, Delta Wing, Fighter, F117, e Torso. A Tabela 6.1 mostra o número de vértices, faces, e tetraedros de cada dado. As imagens geradas variam de 256×256 para 4K×4K pixels. As Figuras 6.2 a 6.8 mostram as imagens geradas para cada um dos dados quando renderizados por nossos algoritmos.

Tabela 6.1: Descrição das massas de dados.

| *Dataset* | **# Verts** | **# Faces** | **# Tets** |
|---|---|---|---|
| **SPX** | 149 K | 1.6 M | 827 K |
| **Blunt Fin** | 41 K | 381 K | 187 K |
| **Oxigen Post** | 109 K | 1 M | 513 K |
| **Delta Wing** | 211 K | 2 M | 1 M |
| **Fighter** | 160 K | 2.8 M | 1.4 M |
| **F-117** | 48 K | 480 M | 240 K |
| **Torso** | 168 K | 2.1 M | 1 M |

## 6.3 Cell-VF-Ray

Na Figura 6.9, mostramos o tempo de execução em segundos do nosso primeiro algoritmo, Cell-VF-Ray, executando no processador Cell BE sem a otimização de *unroll* de *loops*. Foram geradas imagens de 4K×4K para todos os dados. O gráfico mostra o tempo de execução quando o número de SPEs usados na computação aumenta de 1 para 6. Como pode ser observado, o aumento no número de SPEs reduz significativamente o tempo de execução para todos os dados.

A Tabela 6.2 mostra o ganho relativo de desempenho quando executamos nosso algoritmo com a otimização de *unroll* de *loops* nos SPEs, para imagens de 4K×4K. Os tempos de execução ficaram aproximadamente de 20 a 40% mais rápidos do que a versão sem otimização. Este resultado confirma que para *loops* com número finito e conhecido de repetições, o aumento do número de linhas de código compensa no

ganho de desempenho obtido durante a execução do código. A variação do ganho é explicada pelas diferenças de irregularidade dos dados experimentados e pelo ângulo de observação que eles foram amostrados.

A Tabela 6.3 mostra os *speedups* para todas as execuções, já considerando os SPEs otimizados com *unroll* de *loops*. Os *speedups* são relativos à execução sequencial no PPE. Note que, obtemos alguns resultados de *speedups* superlineares, indicando boa escabilidade e indicando que o *pipeline* proposto está constantemente alimentando os SPEs com dados garantindo processando contínuo. O *speedup* foi considerável e compensou o esforço extra de alterar o código para o processador Cell BE.

Tabela 6.2: Ganho de desempenho devido a otimização de *unroll* de *loops*.

| *Dataset* | **Número de SPEs** | | | | | |
|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** |
| **SPX** | 20.6 % | 21.4 % | 22.9 % | 24.4 % | 25.9 % | 27.1 % |
| **Blunt Fin** | 21.6 % | 24.0 % | 27.6 % | 29.3 % | 30.4 % | 31.0 % |
| **Oxigen Post** | 22.8 % | 25.5 % | 27.6 % | 28.5 % | 31.1 % | 33.6 % |
| **Delta Wing** | 20.3 % | 23.0 % | 25.3 % | 26.7 % | 28.4 % | 29.3 % |
| **Fighter** | 24.3 % | 27.9 % | 31.7 % | 33.9 % | 37.5 % | 33.6 % |
| **F117** | 23.4 % | 26.0 % | 28.2 % | 29.1 % | 29.0 % | 28.9 % |
| **Torso** | 18.5 % | 19.4 % | 20.0 % | 20.6 % | 20.7 % | 20.0 % |

Tabela 6.3: *Speedups* de 1 até 6 SPEs do Cell-VF-Ray.

| *Dataset* | **Número de SPEs** | | | | | |
|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** |
| **SPX** | 2.25 | 4.49 | 6.69 | 8.85 | 10.95 | 12.98 |
| **Blunt Fin** | 1.99 | 3.94 | 5.81 | 7.62 | 9.29 | 10.66 |
| **Oxigen Post** | 2.39 | 4.75 | 7.08 | 9.36 | 11.60 | 13.79 |
| **Delta Wing** | 2.33 | 4.60 | 6.79 | 8.88 | 10.86 | 12.70 |
| **Fighter** | 2.86 | 5.54 | 8.10 | 10.33 | 12.48 | 13.24 |
| **F117** | 2.58 | 4.75 | 6.53 | 7.90 | 8.92 | 9.69 |
| **Torso** | 1.87 | 3.69 | 5.39 | 6.99 | 8.46 | 9.75 |

Na Tabela 6.4, mostramos o tempo de execução do algoritmo Cell-VF-Ray, para

todos os dados, quando aumentamos o tamanho da imagem. Para esse experimento, usamos todos os SPEs disponíveis (seis). Como podemos observar na tabela, o tempo de execução aumenta na mesma proporção que o número de pixels aumenta para todos os dados. Esse resultado confirma que nossa solução não está impetrando *overheads* extras quando mais pixels são considerados para a computação.

Tabela 6.4: Tempo de execução por tamanho da imagem do Cell-VF-Ray.

| *Dataset* | **Tamanho da Imagem** | | | | |
|---|---|---|---|---|---|
| | $256^2$ | $512^2$ | $1K^2$ | $2K^2$ | $4K^2$ |
| **SPX** | 0.4 | 1.2 | 4.7 | 18.6 | 73.7 |
| **Blunt Fin** | 0.3 | 1.2 | 4.9 | 19.9 | 80.1 |
| **Oxigen Post** | 0.5 | 2.0 | 8.0 | 32.0 | 128.1 |
| **Delta Wing** | 0.5 | 1.7 | 6.0 | 26.1 | 104.7 |
| **Fighter** | 0.6 | 1.6 | 5.1 | 19.1 | 82.3 |
| **F117** | 0.1 | 0.5 | 2.0 | 8.7 | 44.5 |
| **Torso** | 0.4 | 1.5 | 5.9 | 23.5 | 94.5 |

### 6.3.1 Resultados da Simulação

A Listagem 6.1 apresenta o resultado da simulação do algoritmo Cell-VF-Ray no simulador SystemSim, usando o dado SPX como exemplo.

```
systemsim % mysim spu 0 display statistics
SPU DD3.0
***
Total Cycle count              12023532288
Total Instruction count        6931924874
Total CPI                      1.73
***
Performance Cycle count        12023532288
Performance Instruction count  6931924874 (6162466287)
```

```
Performance CPI                    1.73 (1.95)

Branch instructions                349161386
Branch taken                       243855805
Branch not taken                   105305581

Hint instructions                  120220325
Pipeline flushes                   137156065
SP operations (MADDs=2)            2140150308
DP operations (MADDs=2)            27858412

Contention at LS between Load/Store and Prefetch 309148262

Single cycle                                 3939046057 ( 32.8%)
Dual cycle                                   1111710115 (  9.2%)
Nop cycle                                     324900150 (  2.7%)
Stall due to branch miss                     2414457695 ( 20.1%)
Stall due to prefetch miss                            0 (  0.0%)
Stall due to dependency                      3697531234 ( 30.8%)
Stall due to fp resource conflict                     0 (  0.0%)
Stall due to waiting for hint target          126098254 (  1.0%)
Issue stalls due to pipe hazards               66860190 (  0.6%)
Channel stall cycle                           342928584 (  2.9%)
SPU Initialization cycle                              9 (  0.0%)
-----------------------------------------------------------------
Total cycle                                 12023532288 (100.0%)

The number of used registers are 128, the used ratio is 100.00
```

Listing 6.1: Resultado da simulação do algoritmo Cell-VF-Ray - spx2.off 1K×1K

De forma geral, o simulador nos mostra que a distribuição de trabalho pelo algoritmo Cell-VF-Ray é eficiente, dado que *Channel stall cycle* é de apenas 2.9%, podemos observar que os todos registradores estão sendo usados, já que estamos fazendo uso de *unroll* de *loops*. Há, entretanto, alguns problemas de desempenho detectados como 30.8% devido à *Stall due to dependency*, que ocorre quando em duas instruções consecutivas, a segunda tem dados que precisam ser computados pela primeira e o *dual pipeline* do processador não pode ser usado. Outro dado que destacamos é o *Stall due to branch miss*, pois é difícil eliminar absolutamente todos os *if-then-else* de um código.

## 6.4 Cell-VF-Ray-FB

Na Tabela 6.5 mostramos os *speedups* obtidos pelo algoritmo Cell-VF-Ray-FB para todos os dados, com geração de imagens de 4K×4K, utilizando de 1 até 6 SPEs. Conforme podemos observar nessa tabela, obtemos *speedup* para todos os dados e com aumento do número de SPEs, ocorre o aumento do *speedup*. Além disso, a linearidade do crescimento dos *speedups* mostra que a distribuição do trabalho não está bloqueando a computação nos SPEs, já que com o aumento do número de SPEs, o *speedup* aumenta proporcionalmente.

A Tabela 6.6 mostra o tempo de execução do algoritmo Cell-VF-Ray-FB para diferentes tamanhos de imagem, usando as 6 SPEs disponíveis. Podemos observar nesta que o tempo de execução aumenta para todos os dados, mostrando que nosso código de distribuição de trabalho não está influenciando na produção da imagem final.

A Tabela 6.7 mostra a comparação de *speedups* obtidos pelo algoritmo Cell-VF-Ray-FB com relação aos *speedups* obtidos pelo algoritmo Cell-VF-Ray. Apresentamos nesta tabela apenas a diferença % destes *speedups*, considerando a execução do algo-

Tabela 6.5: *Speedups* de 1 até 6 SPEs do Cell-VF-Ray-FB.

| *Dataset* | **Número de SPEs** | | | | | |
|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** |
| **SPX** | 3.83 | 6.38 | 9.35 | 12.13 | 14.68 | 16.97 |
| **Blunt Fin** | 1.88 | 3.72 | 5.50 | 7.21 | 8.81 | 10.19 |
| **Oxigen Post** | 3.53 | 6.97 | 10.28 | 13.44 | 16.48 | 19.38 |
| **Delta Wing** | 2.96 | 5.79 | 8.43 | 10.89 | 13.12 | 15.04 |
| **Fighter** | 4.09 | 7.84 | 11.02 | 12.43 | 12.42 | 13.56 |
| **F117** | 2.93 | 5.27 | 7.03 | 8.27 | 9.12 | 9.71 |
| **Torso** | 1.89 | 3.70 | 5.38 | 6.93 | 8.35 | 9.61 |

Tabela 6.6: Tempo de execução por tamanho da imagem do Cell-VF-Ray-FB.

| *Dataset* | **Tamanho da Imagem** | | | | |
|---|---|---|---|---|---|
| | $256^2$ | $512^2$ | $1K^2$ | $2K^2$ | $4K^2$ |
| **SPX** | 1.3 | 1.8 | 4.7 | 15.3 | 56.4 |
| **Blunt Fin** | 0.4 | 1.3 | 5.3 | 20.8 | 83.7 |
| **Oxigen Post** | 0.6 | 1.7 | 6.0 | 23.0 | 91.2 |
| **Delta Wing** | 0.6 | 1.7 | 5.9 | 22.4 | 88.4 |
| **Fighter** | 0.9 | 2.3 | 6.6 | 20.7 | 80.2 |
| **F117** | 0.1 | 0.5 | 2.0 | 8.6 | 44.4 |
| **Torso** | 0.5 | 1.6 | 6.1 | 24.0 | 95.8 |

ritmo Cell-VF-Ray como 100%. Valores positivos representam os ganhos de *speedup* do algoritmo Cell-VF-Ray-FB e valores negativos representam as perdas. A irregularidade e a forma dos dados, junto com o ângulo de visualização usado no processamento influenciam diretamente no desempenho do algoritmo Cell-VF-Ray-FB, diferente do algoritmo Cell-VF-Ray, que sempre obtivemos ganhos de desempenho em relação a execução sequencial. Isto ocorre porque o algoritmo Cell-VF-Ray-FB tem o FacesBuffer limitado, já que o LS do SPE é limitado. Portanto, para raios longos, o *overhead* de procurar uma face já processada é maior que o benefício do reuso das faces, pois temos poucas faces para aproveitar ao longo de um raio longo. Outra observação importante é a diferença de resultados quando o número de SPEs cresce. Para alguns dados, como

Tabela 6.7: Comparação de *speedups* entre Cell-VF-Ray e Cell-VF-Ray-FB.

| *Dataset* | **Número de SPEs** | | | | | |
|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** |
| **SPX** | 70.5 % | 42.1 % | 39.8 % | 37.2 % | 34.0 % | 30.7 % |
| **Blunt Fin** | -5.2 % | -5.2 % | -5.2 % | -5.3 % | -5.2 % | -4.3 % |
| **Oxigen Post** | 47.7 % | 46.7 % | 45.1 % | 43.6 % | 42.1 % | 40.5 % |
| **Delta Wing** | 27.3 % | 25.8 % | 24.2 % | 22.6 % | 20.9 % | 18.3 % |
| **Fighter** | 43.0 % | 41.6 % | 35.9 % | 20.3 % | -0.5 % | 2.7 % |
| **F117** | 13.7 % | 11.1 % | 7.69 % | 4.7 % | 2.2 % | 0.2 % |
| **Torso** | 1.1 % | 0.4 % | -0.1 % | -0.9 % | -1.2 % | -1.4 % |

limpamos o FacesBuffer sempre que uma nova face visível é projetada pelo PPE, quanto maior o número de SPEs, menos faces são reaproveitadas e menos eficiente é o algoritmo Cell-VF-Ray-FB.

## 6.5 Comparação com a plataforma Intel

Na Tabela 6.8 mostramos o ganho de desempenho do algoritmo Cell-VF-Ray em relação ao algoritmo VF-Ray sequencial executando na plataforma Intel. O processador utilizado foi um Intel E6750 - Dual Core 2.66 Ghz com 2 GB de memória RAM. A comparação apresenta a diferença % do tempo de execução do algoritmo Cell-VF-Ray no processador Cell BE em relação à execução do algoritmo VF-Ray no processador Intel. A diferença relativa é calculada considerando o tempo de execução no processador Intel como 100% e a diferença é calculada quando executamos apenas no PPE, ou seja, com 0 (zero) SPEs, e variando o número de SPEs de 1 até 6 para todos os dados, com imagens de 4K$\times$4K. Valores menores do que 100% representam resultados em que o algoritmo Cell-VF-Ray executou mais rápido do que o algoritmo VF-Ray. A Tabela 6.9 apresenta a mesma comparação, contudo, feita entre o algoritmos Cell-VF-Ray-FB no

processador Cell BE e o algoritmo VF-Ray na plataforma Intel.

Tabela 6.8: Comparação de tempos de execução, Cell-VF-Ray × VF-Ray (Intel).

| *Dataset* | **Número de SPEs** | | | | | | |
|---|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| **SPX** | 1062.3% | 472.4% | 236.7% | 158.8% | 120.1% | 97.0% | 81.8% |
| **Blunt Fin** | 1171.9% | 589.7% | 298.1% | 201.7% | 153.9% | 126.1% | 109.9% |
| **Oxigen Post** | 1120.2% | 467.8% | 235.9% | 158.9% | 119.7% | 96.6% | 81.2% |
| **Delta Wing** | 646.8% | 277.9% | 140.5% | 95.2% | 72.8% | 59.6% | 50.9% |
| **Fighter** | 998.4% | 349.2% | 180.3% | 123.3% | 96.6% | 80.0% | 75.4% |
| **F117** | 1097.8% | 426.2% | 231.4% | 168.2% | 138.9% | 123.0% | 113.3% |
| **Torso** | 647.4% | 345.9% | 175.7% | 120.1% | 92.6% | 76.5% | 66.4% |

Tabela 6.9: Comparação de tempos de execução, Cell-VF-Ray-FB × VF-Ray (Intel).

| *Dataset* | **Número de SPEs** | | | | | | |
|---|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| **SPX** | 1062.3% | 227.1% | 166.5% | 113.6% | 87.5% | 72.3% | 62.6% |
| **Blunt Fin** | 1171.9% | 622.2% | 314.5% | 212.7% | 162.4% | 132.9% | 114.9% |
| **Oxigen Post** | 1120.2% | 316.7% | 160.8% | 108.9% | 83.3% | 67.9% | 57.8% |
| **Delta Wing** | 646.8% | 218.4% | 111.7% | 76.7% | 59.4% | 49.3% | 43.0% |
| **Fighter** | 998.4% | 244.2% | 127.3% | 90.6% | 80.3% | 80.4% | 73.4% |
| **F117** | 1097.8% | 374.7% | 208.2% | 156.2% | 132.7% | 120.4% | 113.0% |
| **Torso** | 647.4% | 341.9% | 174.9% | 120.2% | 93.4% | 77.5% | 67.3% |

Conforme podemos observar nas Tabelas 6.8 e 6.9, a execução com apenas o PPE é muito pior quando comparada à execução no Intel. Isso ocorre porque atualmente os compiladores da Intel possuem um nível de otimização muito elevado para códigos seriais. Quando usamos os SPEs, observamos que para a maioria dos dados, quando usamos 3 ou mais SPEs, o Cell BE tem melhor desempenho que o processador Intel nos dois algoritmos. Entretanto, podemos observar que dados com número reduzido de tetraedros, como Blunt Fin e F117, não apresentaram melhora no tempo de execução com o uso dos SPEs. Como nesses dados a quantidade de vértices é baixa, é possível fazer bom uso da *cache* do processador Intel. Para dados com quantidade maior de vér-

tices, entretanto, a plataforma Intel sofre com uma quantidade maior de *cache misses* e a implementação no Cell BE tira proveito do gerenciamento em software da movimentação de dados, aproveitando ao máximo a sobreposição dos acessos a dados com computação.

## 6.6 Discussão

Os ganhos no tempo de execução e no *speedup* são ainda modestos quando comparados com o potencial da arquitetura para acelerar aplicações gráficas. O impressionante ganho obtido pela implementação de *raycasting* para malhas regulares no Cell BE proposto por Kim e Jaja em [31] não podem ser reproduzidos para malhas irregulares. A implementação deles obteve enormes acelerações com duas otimizações especiais, aproximação e refinamento, que removeu o *overhead* de latências no acesso a memória. Essas otimizações só podem ser realizadas quando a geometria das malhas é fixa. Além disso, malhas regulares não necessitam acessar a conectividade dos dados, reduzindo as transferências DMA. O *raycasting* para malhas irregulares, por outro lado, é mais desafiador, já que durante a computação do raio, as informações de conectividade precisam ser constantemente consultadas. Fazendo com que o *overhead* de latência no acesso a memória seja um grande obstáculo para o desempenho.

Ainda assim, essa é uma versão preliminar da implementação do *raycasting* de malhas irregulares para o Cell BE. O processador Cell BE é uma nova arquitetura em que todos os níveis de paralelismo precisam ser cuidadosamente explorados, para conquistar todos os benefícios de desempenho. A arquitetura de memória em três níveis, que desacopla o acesso a memória principal da computação e é totalmente controlado por *software*, aumenta a responsabilidade da programação, porém, o pequeno LS dos SPEs

não impuseram uma limitação prática na nossa aplicação. Outra implementação poderia explorar alguns benefícios dessa arquitetura de memória, como transferências de grandes blocos de dados que podem atingir maiores larguras de banda que transferências individuais *cache-line*, e a sobreposição de comunicação e computação programadas por *software*. Nossa abordagem abre a porta para otimizações futuras nas transferências DMA, explorando pré-busca de dados no SPE para diminuir a espera por dados nos SPEs.

Apesar das questões de desempenho, o formato dos dados nos SPEs impuseram outro obstáculo para programação gráfica. Embora os SPEs sejam totalmente compatíveis com padrão IEEE-754 para dados e operações usando ponto-flutuante de precisão dupla, para precisão simples, os resultados não são compatíveis (comportamento diferente em *overflow* e *underflow* e suportam apenas o modo de arredondamento *truncation*).

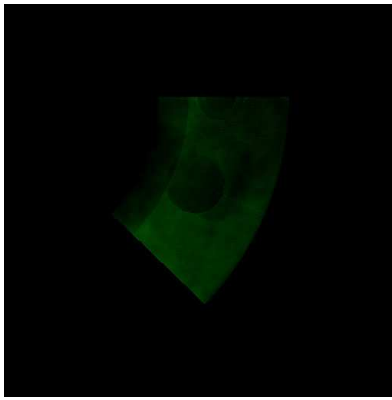

Figura 6.2: SPX.

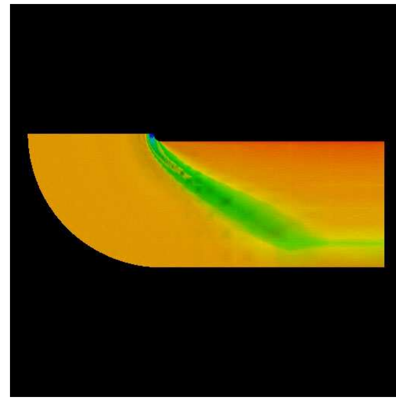

Figura 6.3: Blunt Fin.

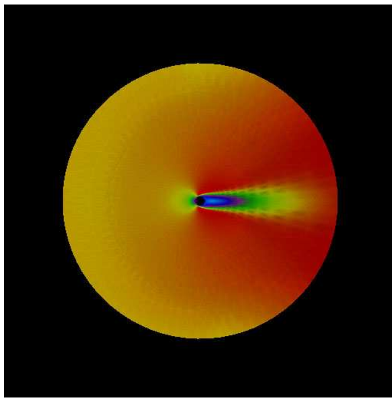

Figura 6.4: Oxigen Post.

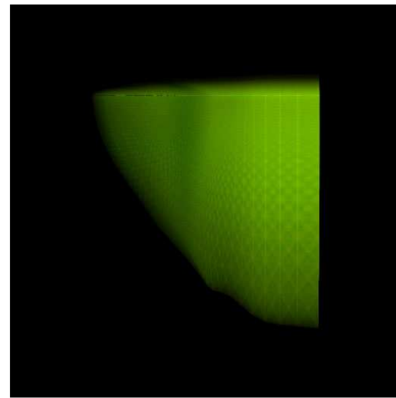

Figura 6.5: Delta Wing.

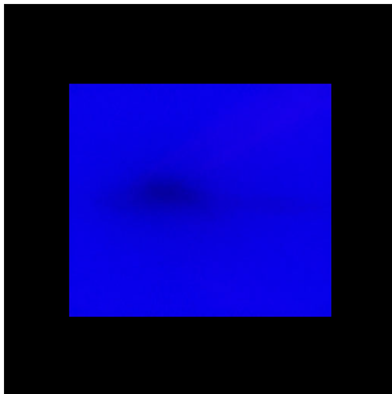

Figura 6.6: Fighter.

Figura 6.7: F-117.

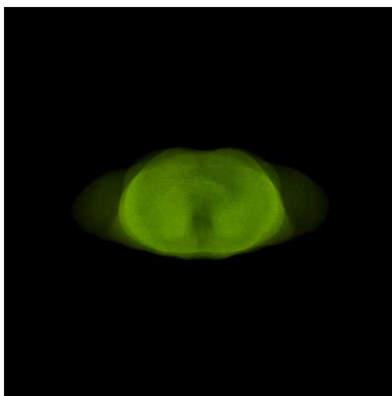

Figura 6.8: Torso.

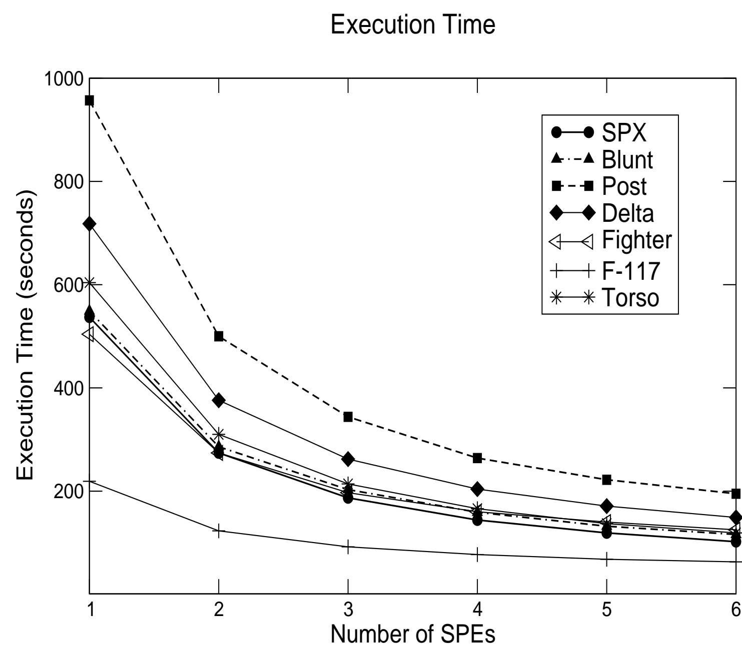


Figura 6.9: Tempo de execução para todos *datasets* - sem *unroll* de *loops*.

# Capítulo 7

# Conclusões

O processador Cell BE oferece uma abordagem de arquitetura inovadora que tem um projeto radicalmente diferente daqueles oferecidos por outros processadores *multicore*. Essa nova arquitetura tem um grande potencial para aumentar a velocidade, o desempenho, e o uso eficiente de energia em muitas aplicações de alto desempenho, incluindo aplicações gráficas.

Entre as características mais marcantes do processador Cell BE estão o seu projeto de núcleos heterogêneos e as hierarquias de *cache* que foram substituidas por uma arquitetura de memória com três níveis controladas por *software*, desacoplando completamente as operações de *load/store* da memória principal da computação. Além disso, a arquitetura fornece recursos de processamento vetoriais SIMD, e um considerável número de registradores.

Por esse motivo, programar aplicações de alto desempenho para Cell BE é desafiador em vários aspectos. Programas que rodam em Cell BE requerem o uso de *multithread*, orquestrando o uso de processamento computacional e transferências de memória, e o uso efetivo dos recursos de SIMD.

O desafio intrínseco de programação paralela para o processador Cell BE é também observado em outras arquiteturas. A rápida evolução das GPUs e CPUs *multicores* patrocina o fim da programação sequencial de processador único, em favor de modelos de programação massivamente paralelos. Indicações desse futuro caminho podem ser observadas na *nVidia's Compute Unified Device Architecture – CUDA*, e no recente anúncio da nova arquitetura da Intel – *Larrabee* – convergindo estratégias de *pipelines* e de projeto da CPU e da GPU.

Nesta dissertação, propomos novas implementações do algoritmo de *raycasting*, VF-Ray, para malhas irregulares que exploram a arquitetura altamente paralela do processador Cell BE. Como o processador Cell BE impõe um modelo de programação diferente, nossa abordagem concentrou-se em reduzir a latência das transferências de memória, transferindo dados de maneira eficiente para as SPEs, e explorando a capacidade SIMD dos núcles de processamento. Nossos algoritmos distribuem a computação do raio para as SPEs, e também vetorizam as operações da integral de iluminação em cada SPE. Nossos resultados experimentais mostraram que podemos atingir bons *speedups* no PS3. Assim, para desvendar o desempenho completo do processandor Cell BE, uma programação cuidadosa é necessária. Todavia, sob adequado desenvolvimento, o processador Cell BE demonstrou potencial para implementação de renderização paralela.

Nossa implementação dos algoritmos de raycasting baseadas em Cell BE produziram duas publicações: [12] e [38]. Como trabalhos futuros, estamos trabalhando em otimizar nossa abordagem, e uma implementação paralela que explore um *cluster* de PS3.

# Referências Bibliográficas


[1] F. Abraham, W. Celes, R Cerqueira, and J.L. Campos. A load-balancing strategy for sort-first distributed rendering. In *17th Brazilian Symposium on Computer Graphics and Image Processing*, pages 292–299, 2004.

[2] Andrew Adinetz, Boris Barladian, Vladimir Galaktionov, Lev Shapiro, and Alexey Voloboy. Abstract physically accurate rendering with coherent ray tracing. In *International Conference on Computer Graphics and Vision, GraphiCon*, 2008.

[3] J Allard and B. Raffin. A shader-based parallel rendering framework. In *Proceedings of IEEE Visualization Conference*, pages 127–134, 2005.

[4] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray tracing on the cell processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[5] Fábio F. Bernardon, Christian A. Pagot, Jo ao Luiz Dihl Comba, and Cláudio T. Silva. GPU-based Tiled Ray Casting using Depth Peeling. *Journal of Graphics Tools*, 11.3:23–29, 2006.

[6] Daniel A. Brokenshire. *Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance*. IBM, Jun 2006.

[7] Hank Childs, Mark Duchaineau, and Kwan-Liu Ma. A scalable, hybrid scheme for volume rendering massive data sets. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 153–162, May 2006.

[8] Alex C. Chow, Gordon C. Fossum, and Daniel A. Brokenshire. *A Programming Example: Large FFT on the Cell Broadband Engine*. IBM, May 2005.

[9] A. Coelho, A. Lopes, C. Bentes, M. de Castro, and Ricardo Farias. Distributed load balancing algorithms for parallel volume rendering on cluster of pcs. In *XXXII Conferencia Latinoamericana de Informática CLEI 2006*, pages 51–58, 2006.

[10] Joao L.D. Comba, Joseph S.B. Mitchell, and Claudio T. Silva. On the convexification of unstructured grids from a scientific visualization perspective. *Scientific Visualization: The Visual Extraction of Knowledge from Data*, pages 17–34, 2006.

[11] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96, 2002.

[12] G. Cox, A. Maximo, C. Bentes, and R. Farias. Irregular grid raycasting implementation on the cell broadband engine. In *21st International Symposium on Computer Architecture and High Performance Computing*, Aceito para publicacao 2009.

[13] E. A. de Oliveira and M. A. Ferreira. Visualizacao da dispersao de efluentes na atmosfera. In *X Simposio Brasileiro de Computacao Grafica e Processamento de Imagens*, 1997.

[14] S. Djurcilov, K. Kim, P.F.J. Lermusiaux, and A. Pang. Visualizing scalar volumetric data with uncertainty. *Computers and Graphics*, 2(26):239–248, April 2002.

[15] J. Dongarra. Top500 supercomputers site. *http://www.top500.org*.

[16] R. Espinha and W. Celes. High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *SIBGRAPI '05: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing*, pages 273–281, 2005.

[17] R. Farias, C. Bentes, A. Coelho, S. Guedes, and L. Goncalves. Work distribution for parallel zsweep algorithm. In *XI Brazilian Symp. on Computer Graphics and Image Processing*, pages 107–114, October 2003.

[18] R. Farias, J. Mitchell, and C. Silva. Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *2000 Volume Visualization Symposium*, pages 91 – 99, October 2000.

[19] Nathan Fout and Kwan-Liu Ma. Transform coding for hardware-accelerated volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(6), November 2007.

[20] J. C. Gonzato and B. Le. Saec. On modeling and rendering ocean scenes. *Journal of Visualisation and Computer Simulation*, 11(1):27–37, January 2000.

[21] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A scalable graphics system for clusters. In *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 129–140, 2001.

[22] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter Kirchner, and James T. Klosowski. Chromium: A stream processing framework for interactive rendering on clusters. In *SIGGRAPH 2002, Computer Graphics Proceedings*, 2002.

[23] IBM. Cell Documentation [online] *http://cell.scei.co.jp/e_download.html*.

[24] IBM. Fedora for PS3 [online] *http://fedoraproject.org/wiki/playstation*.

[25] IBM. SDK Resources [online] *http://www.ibm.com/developerworks/power/cell*.

[26] IBM. *Cell Broadband Engine Software Development Kit Version 2.0*, 2005.

[27] IBM. *Synergistic Processor Unit Instruction Set Architecture - version 1.2*. IBM, 2007.

[28] IBM. *C/C++ Language Extensions for Cell Broadband Engine Architecture*. IBM, 2008.

[29] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[30] A. Kaufman. *Volume Visualization, Encyclopedia of Electrical and Electronics Engineering*. Wiley Publishing, 1997.

[31] J. Kim and J. Jaja. Streaming model based volume ray casting implementation for cell broadband engine. *Scientific Programming*, 17(1-2):173–184, 2009.

[32] B. Lambronici, C. Bentes, L. Drummond, and R. Farias. Dynamic screen division for load balancing the raycasting of irregular data. In *Submitted for publication*, 2009.

[33] Jong Kwan Lee and Timothy S. Newman. Acceleration of opacity correction mechanisms for over-sampled volume ray casting. In *EGPGV '08: Symposium on Parallel Graphics and Visualization*, pages 22–30, 2008.

[34] Kwan-Liu Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *Computer Graphics and Applications, IEEE*, 14(4):59–68, 1994.

[35] S. Marchesin, C. Mongenet, and J. M. Dischler. Dynamic load balancing for parallel volume rendering. In *Eurographics Symposium on Parallel Graphics and Visualization*, Braga, Portugal, 2006.

[36] Stephane Marchesin, Catherine Mongenet, and Jean-Michel Dischler. Oumulti-gpu sort-last volume visualization. In *EGPGV '08: Symposium on Parallel Graphics and Visualization*, pages 1–8, 2008.

[37] R. Marroquim, A. Maximo, R. Farias, and C. Esperança. Volume and Isosurface Rendering with GPU-Accelerated Cell Projection. *Computer Graphics Forum*, 27:24–35, 2008.

[38] A. Maximo, G. Cox, C. Bentes, and R. Farias. Unleashing the power of the playstation 3 to boost graphics programming. *Sibgrapi Tutorial*, 2009.

[39] A. Maximo, S. Ribeiro, C. Bentes, A. Oliveira, and R. Farias. Memory efficient gpu-based ray casting for unstructured volume rendering. In *IEEE/EG Int. Symp. Volume and Point-Based Graph.*, pages 55–62, 2008.

[40] M. Meibner, S. Grimm, W. Straber, J. Packer, and D. Latimer. Parallel volume rendering on a single-chip simd architecture. In *PVG '01: Proceedings of the*

*IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 107–113, 2001.

[41] Steven Edward Molnar. *Image-composition architectures for real-time image generation*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1992.

[42] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 75–84., 1995.

[43] C. Mueller. Hierarchical graphics databases in sort-first. In *PRS '97: Proceedings of the IEEE symposium on Parallel rendering*, pages 49–57., 1997.

[44] C. Muller, M. Strengert, and T. Erl. Optimized volume raycasting for graphics-hardware-based cluster systems. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 59–66, 2006.

[45] Keith O'Conor, Carol O'Sullivan, and Steven Collins. Isosurface extraction on the cell processor. In *Seventh Irish Workshop on Computer Graphics*, pages 57–64, 2006.

[46] Playstation3. PS3's User Manual [online] *http://www.us.playstation.com/support/manuals/ps3*.

[47] C. E. Prakash and A. E. Kaufman. Volume terrain modeling. Technical report, SUNYSB Technical Report, 1997.

[48] T. Funkhouse R. Samant and and K. Li. Parallel rendering with k-way replication.

In *Proc. of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2001.

[49] S. Ribeiro, A. Maximo, C. Bentes, A. Oliveira, and R. Farias. Memory-aware and efficient ray-casting algorithm. In *SIBGRAPI '07: Proceedings of the XX Brazilian Symposium on Computer Graphics and Image Processing*, pages 147–154, 2007.

[50] Daniel Ruijters and Anna Vilanova. Optimizing gpu volume rendering. In *WSCG'06: The 15th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2006.

[51] S. Sakamoto, H. Nishiyama, H. Satoh, S. Shimizu, T. Sanuki, K. Kamijoh, A. Watanabe, and A. Asara. An implementation of the feldkamp algorithm for medical imaging on cell. In *IBM White Paper*, October 2005.

[52] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proc. of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[53] S. R. Santos, C. E. A. Valadao, and M. Dreux. Visualizacao e acompanhamento automatico de sistemas de nuvens. In *IX Simposio Brasileiro de Computacao Grafica e Processamento de Imagens*, 1996.

[54] Matthew Scarpino. *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice Hall, 2008.

[55] Magnus Strengert, Marcelo Magallón, Daniel Weiskopf, Stefan Guthe, and Thomas Ertl. Hierarchical visualization and compression of large volume datasets

using gpu clusters. In *In Eurographics Symposium on Parallel Graphics and Visualization (EGPGV04) (2004*, pages 41–48, 2004.

[56] IBM Systems and Technology Group. *Using Cell Broadband Engine Technology to Improve Molecular Modeling Applications*, 2008.

[57] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3), 2001.

[58] Andrew I. Watson, T. P. Lerico, J. D. Fournier, and E. J. Szoke. The use of d3d when examining tropical cyclones. In *Interactive Symposium on AWIPS*, pages 131–135, June 2002.

[59] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of the 14th IEEE conference on Visualization '03*, pages 333–340, 2003.

[60] Manfred Weiler, Paula N. Mallon, Martin Kraus, and Thomas Ertl. Texture-encoded tetrahedral strips. In *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, pages 71–78, 2004.

[61] Brian Wylie, Constantine Pavlakos, Vasily Lewis, and Ken Moreland. Scalable rendering on pc clusters. *IEEE Comput. Graph. Appl.*, 21(4):62–70, 2001.

[62] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Parallel volume rendering using 2-3 swap image compositing for an arbitrary number of processors. In *Proceedings of Proceedings of IEEE/ACM Supercomputing 2008 Conference*, November 2008.