UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ATILA BOHLKE VASCONCELOS

Modelo de Performance para Código com Desvios de Execução em Hardware Gráfico

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. João Luiz Dihl Comba
Orientador

Prof. Dr. Rui Manuel Ribeiro de Bastos
Co-orientador

Porto Alegre, julho de 2006.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Vasconcelos, Atila Bohlke

Modelo de Performance para Código com Desvios de Execução em Hardware Gráfico / Atila Bohlke Vasconcelos – Porto Alegre: Programa de Pós-Graduação em Computação, 2006.

63 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2006. Orientador: João Luiz Dihl Comba; Co-orientador: Rui Manuel Ribeiro de Bastos.

1.Modelo de Performance. 2.Processamento Gráfico. 3.Hardware Gráfico. I. Comba, João Luiz Dihl. II. Bastos, Rui Emanuel Ribeiro de. III. Título.

Sumário

LISTA DE FIGURAS

LISTA DE TABELAS

Resumo

O advento das unidades de processamento gráfico (GPUs) programáveis forneceram um novo modelo computacional que pode ser utilizado em diversas aplicações. Baseadas em arquitetura de fluxo paralelo, a atual geração de GPUs oferece processadores de vértices e de fragmentos programáveis que podem aumentar drasticamente a performance comparada com soluções implementadas exclusivamente em CPUs. Entretanto obter performance ótima no modelo computacional da GPU, que é complexo e altamente paralelo, com ferramentas de depuração limitadas é uma tarefa difícil e importante. Neste trabalho nós descrevemos uma abordagem simples para avaliar diversas soluções baseadas em GPU para uma dada solução. Ela consiste de um modelo de estimativa de performance que procura reproduzir, dentro de faixas toleráveis de erro, a medida de performance para a unidade de processamento de fragmentos. Nós avaliamos a nossa proposta utilizando as últimas gerações de placas gráficas da NVidia e da ATI usando um conjunto de medidas sintéticas bem como um estudo de caso de uma aplicação em tempo-real.

Palavras-Chave: modelo de performance, processamento gráfico, hardware gráfico

Performance Model for Code with Execution Branches in Graphics Hardware

ABSTRACT

The advent of Graphics Processing Units (GPUs) with programmable shaders brought a new computational model that can be used in several applications. Based on a parallel streaming architecture, current GPU generations offer a vertex and fragment shader that can drastically improve performance if compared to CPU-only solutions. However, obtaining optimal performance in the highly parallel and complex GPU model with limited debugging tools is a challenging and important task. In this work we describe a simple approach to evaluate several GPU alternatives to a given solution. It consists of a performance estimation model that aims to reproduce within acceptable errors the measured performance of the fragment shader. We evaluate our proposal using last generation cards from NVIDIA and ATI using synthetic benchmarks as well as a real-time graphics application case-study.

Keywords: performance model, graphics processing, graphics hardware

1 Introdução

O sub-sistema de processamento gráfico nos computadores pessoais (PCs) modernos tem evoluído em velocidade mais acelerada do que a evolução das CPUs (Unidades Centrais de Processamento). O sub-sistema gráfico é composto por uma placa de expansão (placa de vídeo) que conecta com o sistema através de barramentos PCI, AGP ou PCI-Express. A placa gráfica contém um micro-processador chamado usualmente de GPU (Unidade de Processamento Gráfico) ou VPU (Unidade de Processamento Visual). Devido à natureza intrinsecamente paralela deste tipo de hardware, as GPUs já ultrapassam as CPUs em capacidade de processamento e é esperado que esta vantagem continue aumentando no futuro. Existem aplicações que demandam grande capacidade de processamento, tal qual a GPU pode oferecer.

Algumas classes de aplicações têm mostrado excelente potencial para implementação em GPUs. Isto se deve ao fato dos processadores gráficos terem se mostrado excelentes "*stream processors"* (processadores de fluxo), capazes de processar qualquer informação e não apenas dados gráficos. As GPUs estão cada vez mais perto de tornarem-se pequenos "*super co-processadores"*. Exemplos de aplicações incluem desde a solução de problemas básicos de álgebra linear (GOODNIGHT, 2003), até problemas complexos como a simulação de fluidos (GRIMM, 2004) e Ray Tracing (PURCEL, 2005). Estimar a performance das GPUs é essencial para implementação eficiente destas aplicações.

É importante notar que não se pretende aqui realizar apenas medidas de "*benchmark*". Um benchmark roda diversos programas e gera um relatório de como o hardware responde a execução destes programas. Diversos trabalhos apresentam medidas de performance para soluções em particular; Fathalian (2004) e Hall (2003) reportam os resultados que foram obtidos para multiplicações de matrizes utilizando o processamento na GPU. Trancoso (2005) compara medidas de desempenho realizadas numa CPU e numa GPU. Mas em nenhum dos casos anteriores existe a tentativa de previsão de performance. O que se pretende com este trabalho é simular qual será a performance de uma dada solução para uma determinada configuração.

Estratégias comumente utilizadas para produzir tais estimativas correspondem a um modelo matemático ou construção de um simulador. Para ambas abordagens é necessário conhecimento das características de performance da GPU a ser avaliada. O processo de construção do modelo ou simulador é usualmente feito de forma incremental, começando de uma representação simples e adicionando detalhes sob demanda. Por exemplo, o caso mais simples na GPU é estimar a performance de código executado sem desvios e saltos, sem nenhum acesso a texturas. Quando o código a ser avaliado contém desvios, condicionais ou incondicionais, o modelo necessário para correta previsão da performance se torna mais complexo e necessita de informações

adicionais se comparado com códigos sem desvios. Estes dados adicionais podem estar relacionados com a coerência dos dados sendo processados ou com a freqüência com que os desvios são tomados. Cada tipo de desvio altera o desempenho da arquitetura de maneira diferente e conhecendo como o desempenho da arquitetura varia em cada caso, pode-se construir um modelo para estimar a performance dessa arquitetura sob diferentes cargas de trabalho.

Este trabalho tem por objetivo estimar a performance do hardware gráfico e propor um modelo que seja capaz de auxiliar nesta estimativa. A análise pode ajudar a entender o ganho que determinadas classes de aplicações teriam se fossem efetuadas modificações na maneira em que são implementadas ou mapeadas para a arquitetura da GPU. O modelo deve fornecer informações que permitam avaliar qual a melhor estratégia para se utilizar código com desvios condicionais ou incondicionais. Este modelo deve levar em consideração diversos recursos que o hardware coloca à disposição do programador e como eles podem ser associados para resultar na performance que é obtida. Uma vez proposto o modelo, um simulador pode ser construído para que seja possível verificar a validade do modelo aplicando diversas cargas a cada um dos recursos escolhidos para implementação.

O modelo proposto visa aproximar a performance do processador de fragmentos – um dos componentes internos – de uma GPU. Este modelo é composto de estágios que podem ser agrupados para obtenção do resultado final de estimativa de performance. A composição em estágios tem por finalidade diminuir a complexidade de cada estágio. O modelo apresentado é um modelo de desaceleração, onde uma estimativa de performance máxima é obtida inicialmente e após refinamentos a estimativa é reduzida de acordo com a carga a que cada recurso do hardware está submetido. Os estágios são aplicados seqüencialmente e cada estágio entrega como resultado uma estimativa de performance parcial até o último estágio ser utilizado. Os estágios do modelo podem tomar como dados de entrada parâmetros relativos ao hardware e/ou relativos aos dados sendo processados.

Um simulador composto por estágios foi implementado tendo como base o modelo proposto. O primeiro estágio gera como saída uma estimativa de performance máxima e toma como parâmetros de entrada dados relativos ao hardware e à carga à qual ele está submetido:

- número de pipelines
- freqüência de operação do processador
- número de ciclos dos programas sendo executados
- quantidade de fragmentos sendo processados

O segundo estágio do simulador é responsável por desaceleração devido a perda de performance que está associada com a coerência dos dados sendo processados. Para isto, o estágio toma como parâmetros de entrada:

- estimativa de performance do estágio anterior
- tamanho dos triângulos desenhados
- tamanho do bloco de coerência das tomadas de desvio

O terceiro estágio do simulador é responsável pela introdução de desaceleração devido aos acessos de memória, representados no hardware pelo acessos de leitura a texturas. Este estágio toma como parâmetros de entrada:

- estimativa de performance do estágio anterior
- tamanho das texturas acessadas
- formato (ou bytes por texel) das texturas

A validação dos dados previstos pelos estágios do simulador foi realizada comparando resultados de performance com a execução de aplicações simples em situações sintéticas, onde apenas um parâmetro foi variado de cada vez. Em seguida, o simulador foi verificado com a execução de uma aplicação complexa: um simulador massa-mola, implementado em GPU. Foi estabelecido que o erro relativo médio máximo admitido não ultrapassaria 10% em cada estágio do simulador durante as medidas sintéticas. Para o simulador com três estágios seqüenciais que apresentamos, compondo o erro relativo de cada estágio espera-se que o erro final do simulador não ultrapasse 33%, isto é:

$$\underbrace{\underbrace{[\underbrace{(X * 1{,}1}_{\text{estágio 1}})}_{\text{estágio 2}} * 1{,}1]}_{\text{estágio 3}} * 1{,}1 = 1{,}33 * X$$

dos quais $1{,}0 * X$ é o sinal e $0{,}33 * X$ é o erro.

As estimativas do simulador comparadas com medidas obtidas com experimentos sintéticos executados na GPU indicam um erro relativo dentro do estabelecido como aceitável. Para simulações que não levam em conta acessos a memória da placa de vídeo, o simulador consegue reportar estimativas com erros menores que 2%. No caso do experimento do simulador massa-mola, utilizado para verificar o simulador, o erro relativo não ultrapassou 10%.

Esse texto está organizado em 6 capítulos que podem ser divididos em dois grupos. A parte inicial procura colocar o leitor no contexto da utilização do hardware gráfico. A segunda parte aborda o tema central do trabalho que é a proposição de um modelo de performance e avaliação de um simulador baseado no modelo proposto.

No capítulo 2 o leitor é introduzido ao hardware gráfico e seu funcionamento, como é utilizado para geração de imagens e como pode ser programado. São apresentadas APIs de programação que fazem a interface entre a aplicação que roda na CPU e os programas que são executados de fato pelos processadores do hardware gráfico. No capítulo 3 é apresentada a técnica utilizada para coleta de dados de desempenho. O capítulo 4 propõe o nosso modelo de performance do hardware gráfico parametrizado por recursos que escolhemos como significativos na descrição da arquitetura do hardware gráfico. O modelo recebe como entrada os parâmetros escolhidos e combina-os para entregar uma estimativa de performance dentro do limite de erro relativo estabelecido. No capítulo 5, o simulador que implementa o modelo de performance proposto é avaliado e verificado; são mostradas medidas de performance usando situações simples e complexas. O capítulo 6 encerra o trabalho, avaliando a eficácia do modelo e do simulador com relação a dados reais e propõe trabalhos futuros.

2 Hardware Gráfico

Este capítulo apresenta uma descrição do hardware gráfico. Inicia com uma visão abrangente do hardware usado como base neste trabalho e parte da discussão baseia-se no capítulo sobre a arquitetura da GeForce 6800 discutido em "GPU Gems 2" (KILGARIFF, 2005). Descreve a associação entre CPU e GPU e mostra os diversos estágios do pipeline gráfico e a arquitetura das modernas GPUs. Explica a programação dos processadores da GPU e faz uma breve menção sobre as linguagens e APIs empregadas para tal.

O processamento gráfico é decomposto em diversas atividades que podem ser representadas por estágios funcionais num pipeline (semelhante a uma linha de montagem): processamento da aplicação, processamento de geometria, rasterização e renderização. Cada um destes estágios funcionais é implementado por um ou mais estágios correspondentes no hardware gráfico (BOUATOUCH, 2002). A aplicação (responsável pelo envio de comandos a serem executados pelos demais estágios do pipeline gráfico) roda no processador central da estação de trabalho (CPU). Os demais estágios são implementados por uma unidade de processamento especializada chamada GPU.

A GPU é composta de estágios responsáveis pelo processamento de vértices, estágios que realizam rasterização das primitivas (pontos, linhas e planos) formadas pela geometria dos vértices e estágios que processam os fragmentos rasterizados. Estes estágios podem ser programados (MARK, 2001) através de comandos enviados pela aplicação para realizar determinada operação sobre os dados que estão passando por estes estágios. Estes comandos são gerados pela aplicação que executa na CPU do sistema; portanto existe uma comunicação entre a CPU e a GPU, para o envio dos comandos e dos dados a serem processados.

2.1 Arquitetura da GPU

O processamento gráfico é composto de diversos procedimentos que devem ser realizados sobre dados para a geração de uma imagem. Estes procedimentos são realizados de maneira seqüencial, de modo similar a uma linha de montagem. No início desta linha de montagem está a CPU que envia ao sub-sistema gráfico os dados e os comandos que vão compor a imagem. Na extremidade final da linha de montagem está o display gráfico (monitor) do sistema. Esta linha de montagem é representada pela placa de vídeo do computador (o sub-sistema gráfico). A placa de vídeo é composta por um conjunto de processadores que coletivamente são chamados de GPU (Graphics Processor Unit) e bancos de memória onde dados temporários são armazenados.

A placa de vídeo, sendo um sub-sistema do computador, interage com o resto do hardware por uma interface padronizada, como os barramentos PCI, AGP ou PCI-Express. É por esta interface que comandos e dados a serem processados pela GPU ou armazenados na memória da placa de vídeo trafegam. Devido à grande quantidade de dados e comandos a serem repassados para a GPU, é importante que esta interface não seja um gargalo para o sistema. Por isto, têm sido desenvolvidos barramentos cada vez mais rápidos, culminando atualmente com o PCI-Express que provê uma largura de banda de 8 GB/s entre a CPU e a GPU – figura 2.1.

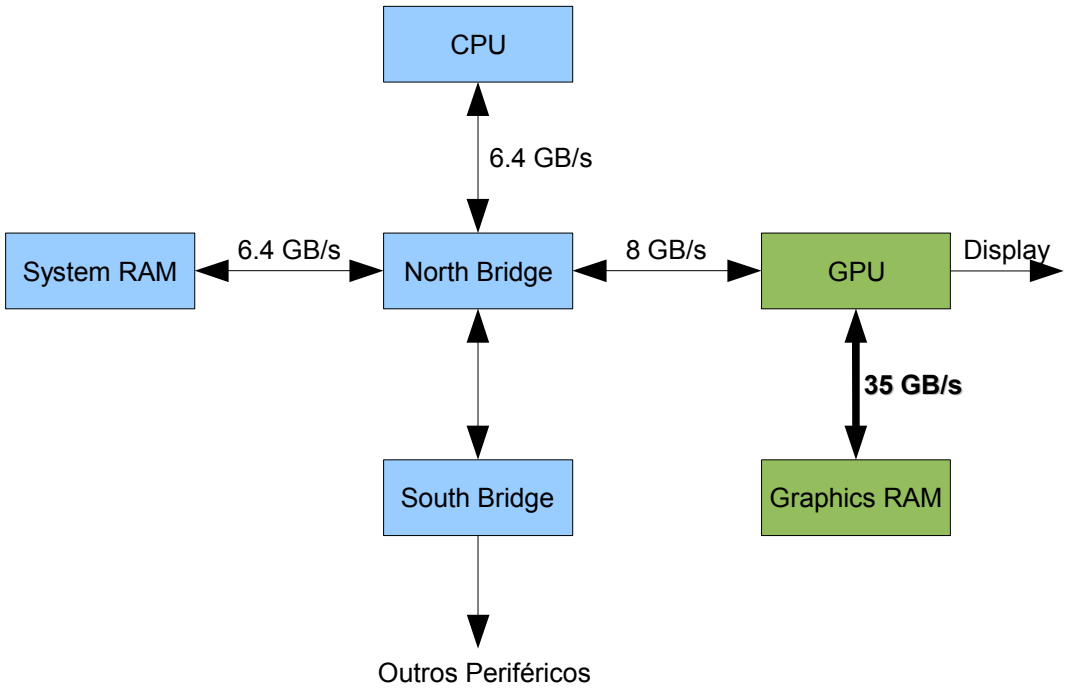

Figura 2.1: Diagrama de blocos da arquitetura simplificada de um PC (extraído de GPU Gems 2)

A GPU é organizada numa linha de montagem (pipeline), similar à linha de montagem do processamento gráfico. A GPU contém processadores que se encarregam do processamento da geometria da cena a ser composta, unidades de rasterização, que transformam a geometria da cena numa imagem plana em duas dimensões composta por pequenos fragmentos de imagens, processadores que operam sobre os fragmentos e um sistema que apresenta o resultado no display gráfico.

No início desta linha de montagem – figura 2.2 – estão os processadores de vértices, que são os primeiros a receber os comandos e dados provenientes da CPU. A figura 2.3 apresenta a arquitetura dos processadores de vértices. Os dados recebidos por estes processadores são vértices que contêm propriedades, tais como posição espacial, cor, etc. Os comandos recebidos pelos processadores de vértices contêm instruções de como agrupar os vértices recebidos, de forma a gerar o que é chamado de primitiva. Uma primitiva é uma figura geométrica espacial simples, em geral linhas e triângulos. Primitivas são combinadas para formar figuras mais complexas a serem exibidas na imagem final. A decomposição dos objetos da imagem em primitivas tem a finalidade de facilitar os processamento em cada estágio do pipeline gráfico, de forma que as

operações enviadas pela aplicação (a partir da CPU) sempre sejam realizadas da maneira mais rápida possível.

As primitivas montadas e manipuladas pelos processadores de vértices são enviadas para as unidades de rasterização. Um rasterizador tem a função de analisar a primitiva que recebe, verificar as propriedades dos vértices que a compõe e então interpolar valores entre estes vértices. O rasterizador obtém isto, tomando o espaço entre cada vértice e subdividindo-o em pequenos pedaços, os fragmentos. Idealmente cada fragmento deveria ser interpretado como um ponto no espaço, mas é tratado como uma pequena área associada ao ponto. Para cada fragmento, certos valores indicados pelos vértices são interpolados, tais como a cor e a posição espacial entre os extremos dados pelos vértices da primitiva.

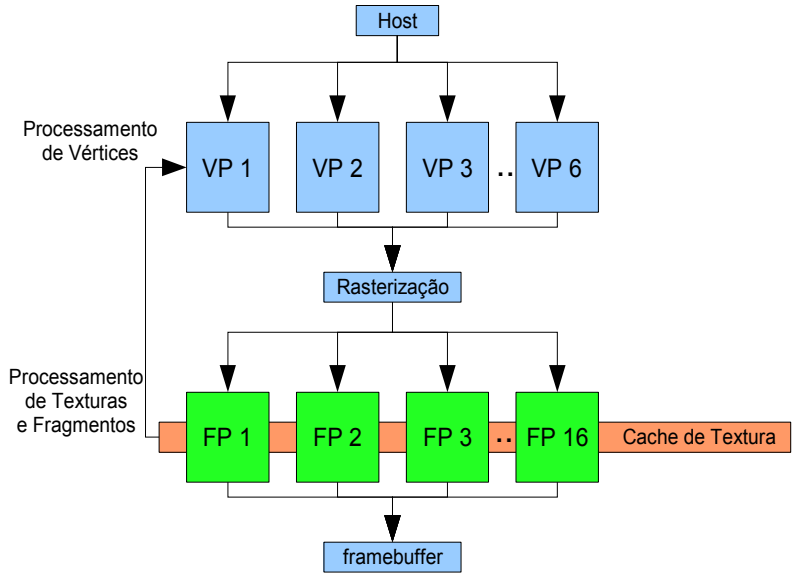

Figura 2.2: Diagrama de blocos simplificado da GPU (extraído de GPU Gems 2)

A última parte desta linha de montagem da GPU, antes da imagem final ser apresentada, trata do processamento que deve ser aplicado sobre os fragmentos provenientes do rasterizador. Esta parte é composta por um número de processadores que é maior do que o número de processadores de vértices, pois o número de fragmentos gerados por cada primitiva em geral supera em muitas ordens de grandeza o número total de primitivas que compõem uma imagem. Estes processadores são chamados de processadores de fragmentos. A figura 2.4 mostra a arquitetura dos processadores de fragmentos. Um processador de fragmentos tem a função de aplicar sobre cada fragmento recebido um conjunto de instruções que lhe foi enviado pela aplicação (ou seja um programa a ser executado para cada fragmento. Os fragmentos são independentes uns dos outros, o que permite os processadores de fragmentos trabalharem de forma coordenada e maximizar o número de fragmentos processados em paralelo. Conforme os fragmentos vão terminando seu processamento, os resultados vão sendo depositados numa área da memória da placa de vídeo chamada de framebuffer. Os dados contidos nessa memória são chamados de pixels. Cada pixel no framebuffer corresponde a um elemento de imagem que compõe o display (monitor) conectado à placa de vídeo.

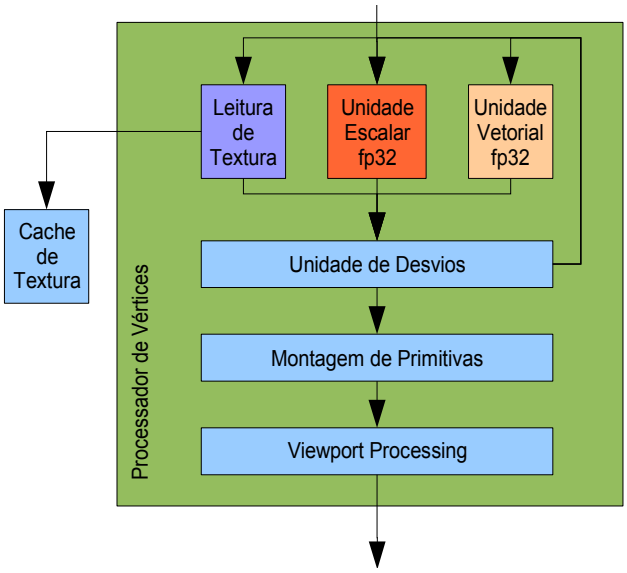

Figura 2.3: O processador de vértices (extraído de GPU Gems 2)

2.2 Processador de Fragmentos

O foco deste trabalho foi centrado no processador de fragmentos, uma parte da linha de montagem do processamento gráfico. Nesta seção vamos aprofundar o conhecimento sobre o processador de fragmentos e algumas de suas estruturas internas para termos um clara definição do que estará sendo tratado nos capítulos a seguir. Começamos por uma breve descrição da evolução dos processadores de fragmentos para então analisar como o processador de fragmentos é implementado nas gerações mais recentes das placas de vídeo.

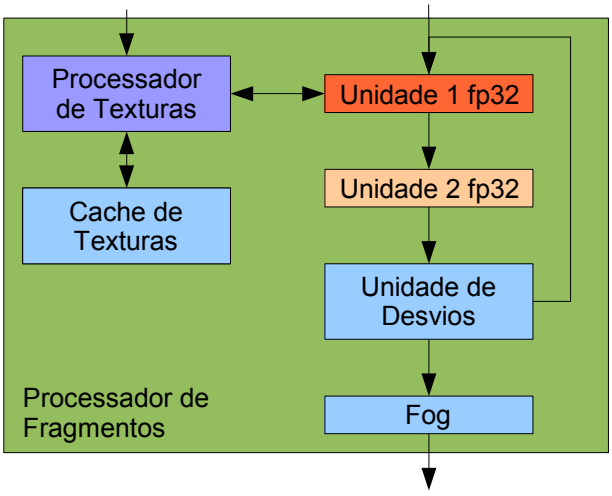

Figura 2.4: Processador de Fragmentos (extraído de GPU Gems 2)

O processador de fragmentos é um dos vários estágios do processamento gráfico. Este processador é composto por pipelines internos. Dados entram por uma extremidade do processador e, após serem manipulados, saem pela outra extremidade do processador. O processador de fragmentos pode ser entendido como sendo composto por um ou mais pipelines, com diversos estágios que realizam diferentes tarefas.

Inicialmente estes estágios internos do processador de fragmentos eram simples e realizavam tarefas simples, não havendo a necessidade de programação. Eventualmente alguma configuração para alterar certos estados dos estágios dos pipelines do processador de fragmentos. Os primeiros processadores de fragmentos eram apenas uma unidade de aplicação de textura aos fragmentos que passavam por eles. A aplicação de uma textura consiste na leitura de um mapa de cores, a textura, e a correta aplicação da cor a um dado fragmento de acordo com a sua posição. Nessa situação, apenas a textura a ser aplicada necessitava ser configurada, uma vez que a tarefa a ser realizada era sempre a mesma: ler a cor de uma posição na textura e transferir esta cor para o fragmento.

Com a evolução do hardware, tarefas mais complexas começaram a ser disponibilizadas nos processadores de fragmentos. A aplicação simultânea de mais de uma textura, por exemplo, exigiu a leitura de dois valores de cor para cada fragmento, a combinação destes valores e a escrita do valor resultante para o fragmento. Isto criou a necessidade de mais informações além da simples menção das texturas a serem utilizadas, pois cada textura pode ter um peso diferente na composição da cor resultante. Mais dados são necessários como entrada e mais estágios no pipeline precisam ser utilizados. Com o tempo, operações aritméticas também começaram a se tornar desejáveis. Estas novas operações exigiram novos estágios no pipeline do processador de fragmentos. Com mais tarefas a serem realizadas, a simples configuração de cada estágio começou a se mostrar uma limitação para a completa utilização do pipeline de fragmentos.

Os estágios de um pipeline de um processador de fragmentos atualmente compreendem desde unidades de leitura de texturas até unidades de tomadas de desvios condicionais, passando por unidades aritméticas. O conjunto destas unidades pode ser programado para realizar uma determinada tarefa. Para cada fragmento ter um programa totalmente aplicado sobre si, podem ser necessárias diversas passadas pelas unidades do pipeline de fragmentos, onde em cada passada as unidades do pipeline estarão configuradas de maneira diferente, correspondendo às instruções do programa. Um programa longo exige diversas passadas pelo pipeline, enquanto um programa curto pode exigir poucas passadas. Este número de passadas necessárias está intimamente ligado ao programa que está em execução e à forma como os diversos estágios do pipeline estão organizados. Dependendo da organização do pipeline, as instruções que compõem um programa podem ser agrupadas e executadas de maneiras diferentes. Em geral a organização do pipeline muda para cada modelo de GPU.

Os pipelines do processador de fragmentos da NVidia obedecem a estruturas similares: duas unidades de operações aritméticas e uma unidade de tomada de desvios (BAUMANN, 2006). A primeira unidade aritmética pode realizar operações de leitura de textura no lugar de uma operação matemática. Um fragmento pode, portanto, ter executada uma instrução de textura, uma instrução aritmética e uma instrução de desvio ou duas instruções aritméticas e uma instrução de desvio. Se mais acessos a texturas ou mais instruções aritméticas são necessárias do que disponível em unidades no pipeline,

então um fragmento pode tem que entrar novamente no pipeline, desta vez com as novas instruções configuradas em cada estágio.

A ATI utiliza uma abordagem diferente para os pipelines do processador de fragmentos (TRIOLET, 2006). O processador de fragmentos das GPUs da ATI têm pipelines especializados na leitura de texturas e outros pipelines especializados em operações aritméticas e desvios de fluxo de execução. Os pipelines de leitura de textura realizam apenas esta tarefa e após um fragmento ter passado por ele, deve ser encaminhado a um pipeline de processamento aritmético para realizar a as operações matemáticas que eventualmente necessite. Os pipelines de processamento aritmético têm duas unidades que realizam operações matemáticas e uma unidade de desvios. Destas duas unidades aritméticas, a primeira é especializada em operações de soma, não estando apta a realizar outro tipo de operação. A segunda unidade aritmética pode realizar tanto instruções de soma quanto as demais instruções disponibilizadas pelo hardware.

A configuração dos estágios do pipeline é feita via um agrupamento de instruções que é enviado para o pipeline e vai reconfigurando os estágios seqüêncialmente. Este agrupamento de instruções é chamado de VLIW (PATTERSON, 1996), do inglês Very Long Instruction Word (Palavra de Instrução Muito Longa). A VLIW é um agrupamento de instruções que são endereçadas aos estágios aos quais elas devem configurar. A responsabilidade da montagem mais adequada desta VLIW é do compilador, que tem conhecimento detalhado sobre o hardware para o qual ele está gerando o código. Os modelos do hardware para os quais VLIWs são construídas, são diferentes entre si. Por mais sutil que sejam estas diferenças, elas devem ser levadas em conta pelo compilador que está gerando as VLIWs. Por isso um programa otimizado para um determinado hardware jamais irá executar com a mesma performance em outro (se eventualmente puder executar). Isso implica recompilar um determinado programa para cada hardware no qual ele vai ser executado. Esta recompilação pode ocorrer antecipadamente ou na hora de execução. Existem aplicações que podem examinar o programa a ser executado antecipadamente e prever quantas VLIWs serão formadas de acordo com as características do pipeline em questão.

Para cada VLIW formada é necessária uma passada (ciclo) do fragmento a ser processado pelo pipeline. Em situações especiais, quando são usadas instruções complexas pode ser necessária mais de uma passada (ou mais de uma VLIW) para a completa execução de uma única instrução. Instruções de desvio são um exemplo de instruções que podem necessitar de mais de uma passada pelo pipeline – tabela . Na primeira passada, por exemplo, podem ser decididos quais fragmentos vão tomar o desvio; na segunda passada, pode ser decidido o salto necessário para a tomada do desvio. A correspondência entre o número de ciclos necessários para a execução de uma instrução sobre o fragmento não precisa ser constante entre os modelos diferentes de hardware, pois cada modelo tem suas particularidades.

Tabela 2.1: Custo das instruções de desvio na GPU G70. Cada instrução de controle de fluxo adicionada a um programa de fragmentos incorre numa sobrecarga de dois ciclos ao programa (extraído de GPU Gems 2).

Instrução	***Custo (ciclos)***
if/endif	4
if/else/endif	6
call	2
ret	2
loop/endloop	4

A maneira como uma VLIW é montada pode dar pistas sobre as particularidades internas dos pipelines, uma vez que cada instrução deve corresponder ao estágio a que ela se destina. Uma destas particularidades diz respeito à maneira paralela como os dados são tratados dentro dos estágios do pipeline. Os dados sobre os quais o processador de fragmentos trabalha são fragmentos. Fragmentos possuem 4 elementos de cor: os canais Vermelho ("*Red*"), Verde ("*Green*"), Azul ("*Blue*") e Alfa, indicam a combinação de cores básicas que deve ser utilizada para obter a cor resultante e mais o percentual de transparência do fragmento. Estes quatro elementos podem ser tratados como elementos de um vetor de quatro posições, uma vez que todos os fragmentos possuem estes mesmos elementos. Fragmentos possuem também coordenadas, em geral bi-dimensionais, que indicam sua posição relativa na textura que deve ser aplicada, sendo por isso facilmente tratadas como vetores. Uma vez que os fragmentos podem ser descritos por vetores é de se esperar que os estágios do pipeline de fragmentos sejam especializados no tratamento de vetores – como de fato o são.

A taxionomia de Flynn (FLYNN, 1966) é usada para classificar como operam os processadores da GPU. Flynn classifica processadores em 3 tipos: processadores SIMD, MIMD e SISD. Processadores SISD executam uma única instrução sobre um único dado. Processadores SIMD se caracterizam por executar uma mesma instrução sobre um grupo de dados ao mesmo tempo. Processadores MIMD (Multiple Intructions Multiple Data) se caracterizam por executar diferentes instruções sobre diferentes dados. As unidades de execução de um pipeline de fragmentos podem trabalhar de uma maneira SIMD sobre os vetores que descrevem os fragmentos. Isto é, diversos pipelines podem estar aplicando a mesma instrução a diversos fragmentos.

3 Medição de Performance

A análise de performance de sistemas computacionais pode ser entendida como uma combinação de medidas e interpretações sobre diversas características do sistema, como performance, velocidade de comunicação, tamanho dos dados, etc. (JAIN, 1991). Devido à complexidade de alguns sistemas, em muitos casos a análise concentra-se em apenas um subconjunto do sistema inteiro. Por exemplo, pode-se estar interessado apenas na performance do subsistema gráfico de um computador. Entretanto os componentes de um sistema computacional podem interagir de uma maneira muito complexa e muitas vezes de maneira imprevisível. Para a análise ser efetiva, é necessário desenvolver técnicas que perturbem o mínimo possível o sistema a ser analisado.

Os objetivos da análise de um sistema computacional dependem da situação específica, dos interesses e da habilidade do analista (LILJA, 2000). Entretanto é possível identificar alguns objetivos típicos:

- Comparação de alternativas: identificar a mais eficaz entre várias configurações.
- Determinação do impacto de uma funcionalidade: medir o impacto na performance quando uma funcionalidade é adicionada ou removida.
- Ajuste fino de sistemas: qual conjunto de configurações resulta na melhor performance.
- Identificação de performance relativa: identificar a performance relativa a dados obtidos de um histórico.
- Detecção de problemas de performance: descobrir por que um sistema não está desenvolvendo a performance esperada.
- Expectativas de performance: indicar o que pode ser esperado de um dado sistema.

Um objetivo comum a todos estes listados acima é caracterizar como a performance do sistema (ou sub-sistema) muda quando certos parâmetros são variados. Quando deparado com problemas de análise de performance, existem três técnicas fundamentais que podem ser empregadas:

- Medida em um sistema existente
- Simulação do sistema

- Modelagem analítica

As medidas realizadas em um sistema existente levam aos melhores resultados da análise de performance. Dado que existam as ferramentas necessárias para a realização das medidas, nenhuma simplificação sobre o sistema precisa ser feita. Isto também torna os resultados mais confiáveis. Realizar medidas em sistemas reais não é muito flexível, uma vez que estas medidas proporcionam informação apenas sobre o sistema medido. Por exemplo, em muitos sistemas não é possível variar a velocidade do subsistema de memória.

A simulação é um instrumento para modelar as características importantes de um sistema. Projetada de forma a ser facilmente modificada, permite estudar o impacto de modificações em vários componentes do sistema simulado. O custo da simulação depende da complexidade do sistema sendo modelado e do grau de detalhamento que está sendo incorporado ao simulador. A primeira limitação na construção de um simulador é a impossibilidade de se modelar todos os detalhes microscópicos do sistema sendo estudado. Simplificações devem ser assumidas para que seja viável a construção do simulador, ciente de que estas devem afetar a precisão e a credibilidade da análise realizada.

A modelagem analítica é uma descrição matemática do sistema. Comparado com as técnicas de medição e de simulação, a modelagem analítica é muito mais complexa de se realizar. É comum optar por representações analíticas mais simples, como aproximações lineares, mas isso faz com que a modelagem seja menos precisa. Um modelo analítico simples permite uma compreensão rápida do sistema, concentrando as medidas em um dado mais específico. Ele ainda ajuda a confirmar os resultados provenientes de um simulador ou de medidas realizadas.

Os critérios para a escolha de uma técnica de análise de performance estão resumidos na tabela 3.1.

Tabela 3.1: Comparação das técnicas de análise de performance

Característica	***Medida***	***Simulação***	***Modelo Analítico***
Flexibilidade	Baixa	Alta	Alta
Custo	Alto	Médio	Baixo
Credibilidade	Alta	Média	Baixa
Precisão	Alta	Média	Baixa

Este trabalho tem como meta a proposição de um modelo para previsão de performance da GPU. Para isto será construído um simulador com base no modelo e seus resultados comparados com medidas realizadas em equipamentos de referência. Deve-se, portanto, conhecer e saber aplicar técnicas e ferramentas que permitam que medidas sejam realizadas e em seguida comparadas com previsões que o simulador possa fornecer.

Neste capítulo são apresentadas algumas ferramentas que podem ser utilizadas para medição de performance do hardware gráfico. A escolha do método utilizado para a medição da performance em um hardware de referência também é apresentada, bem como a métrica utilizada para medição de performance e quais parâmetros foram escolhidos para avaliação.

3.1 Métrica de Performance

Antes de ser entendido qualquer aspecto da performance de um sistema computacional, deve-se determinar quais aspectos são importantes e úteis para a medida (LILJA, 2000). As características básicas que tipicamente se mede em um sistema computacional são:

- contagem de quantas vezes um evento ocorre
- a duração de algum intervalo de tempo
- característica de um parâmetro de entrada

Por exemplo, pode-se estar interessado em quantas vezes um processador inicia uma requisição de entrada e saída, ou quanto tempo cada uma dessas requisições demora. Para este tipo de dado, utiliza-se diretamente o valor medido. Este valor é chamado de métrica de performance. Se o valor interessado é o tempo, contagem ou tamanho de um valor medido, pode-se utilizar o valor diretamente como a métrica de performance. Seguidamente utiliza-se a contagem normalizada de eventos, por exemplo, a uma base comum de tempo para prover uma métrica de velocidade, tal como operações executadas por segundo. Este tipo de métrica é chamado de métrica de taxa. Em geral a métrica de taxa é calculada dividindo a contagem do número de eventos que ocorrem em um dado intervalo de tempo pelo intervalo de tempo em que eles ocorrem. Uma vez que uma métrica de taxa é normalizada para uma base de tempo comum, é razoável utilizar métricas de taxa para comparações.

Uma métrica escolhida para medição de performance de hardware gráfico é o número de quadros que podem ser gerados em um segundo. Um quadro é todo trabalho necessário para exibir uma imagem na tela. Esta métrica recebe o nome de Quadros Por Segundo (abreviada na sua forma em inglês, FPS – Frames Per Second).

Muitas vezes pode-se medir um valor com grandezas inversamente proporcionais. É o caso do FPS. Ao contrário de contar quantos quadros são completados em um segundo, pode-se medir o tempo necessário para o desenho de um quadro. Sabendo o tempo que um quadro leva para ser desenhado, é uma questão de inverter a razão para saber quantos quadros podem ser desenhados em um segundo:

$$\mathrm{FPS} = \frac{1}{tempo\ de\ 1\ quadro} \tag{3.1}$$

Embora FPS seja uma métrica simples de entender, pode levar a resultados não intuitivos, se for utilizada para comparar performance de processadores executando com diferentes freqüências. Para evitar este problema, normalizam-se as medidas pela freqüência dos processadores e obtém-se FPC (do inglês Frames Per Clock):

$$\mathrm{FPC} = \frac{FPS}{freqüência\ do\ processador} \tag{3.2}$$

Para medir corretamente o tempo necessário para o desenho de um quadro, deve-se ter em mente que durante a sua composição, um quadro passa por diversos estágios até que esteja finalizado. No caso do hardware gráfico, alguns destes estágios podem estar executando em processadores distintos. A seguir tem-se uma breve discussão sobre os estágios que estão envolvidos na composição de um quadro. Esta discussão é necessária para que se torne claro como as medidas de tempo estarão sendo implementadas, em qual estágio os dados estão sendo coletados e como essa coleta é realizada.

3.1.1 Estágios do processamento gráfico

O processamento gráfico é realizado em diversos estágios (BOUATOUCH, 2002). A geração de uma imagem é feita através de comandos que processam formas geométricas primitivas. Em geral o triângulo é utilizado como primitiva básica e agrupado para compor primitivas geométricas mais complexas. Comandos de desenho informam ao hardware gráfico características destes triângulos, tais como posição dos vértices, cores, etc. Estes comandos de desenho são gerados e enviados para o hardware gráfico por uma aplicação. O processamento destes comandos passa por diversos estágios de processamento – figura 3.1 – alguns rodando na CPU do sistema, outros rodando na GPU. Para a obtenção de medidas de performance, deve-se realizar a amostragem em lugares adequados destes estágios – figura 3.4. Cada um destes estágios pode interferir com o trabalho dos demais, na medida em que podem realizar otimizações, podem estar sobrecarregados (um gargalo no sistema) ou podem simplesmente não existir em um dado sistema. Processar uma primitiva pode levar um determinado tempo, enquanto que processar 2 ou 3 primitivas pode levar um tempo apenas ligeiramente maior – e não 2 ou 3 vezes maior– devido a um agrupamento de instruções.

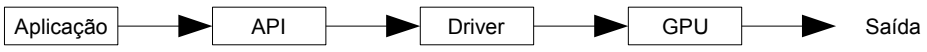

Figura 3.1: Estágios do Processamento Gráfico

O estágio inicial do processamento reside na aplicação, que utiliza comandos de desenho disponibilizados pela API – figura 3.3. A aplicação controla a cena, lida com interações com o usuário e determina como o desenho será efetuado. Todo o trabalho é especificado por uma seqüência de comandos de desenho que é enviada para a API. Essa seqüência de comandos de desenho é independente do hardware utilizado.

A API converte essa seqüência de comandos em um formato independente do hardware, mas pode introduzir algumas otimizações (sem conhecimento sobre o hardware que será utilizado), realizando agrupamentos, ou troca de ordem de instruções. Depois de reagrupadas e reordenadas as chamadas de função da API, as instruções de desenho são repassadas para o driver, que é diferente para cada hardware. A API

utilizada neste trabalho é a DirectX (DIRECTX) desenvolvida pela Microsoft e acompanha os sistemas operacionais Windows.

O driver do sistema operacional conhece o hardware (GPU) disponível e faz otimizações adicionais, desta vez mais agressivas que as anteriores, para tirar o máximo proveito da GPU. O driver pode, por exemplo, otimizar a utilização de registradores que um programa referencia. Também deve se considerar que o driver retorna o controle da execução para a API imediatamente após ter enviado as instruções de desenho para a GPU, sem esperar pela finalização do processo de desenho, para evitar que a CPU fique ociosa enquanto a GPU trabalha. Isto dificulta a medida do tempo gasto pela GPU na execução das instruções.

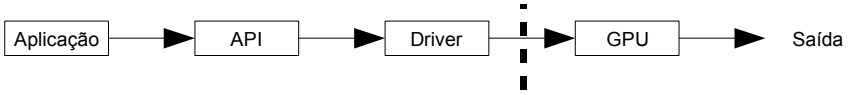

Figura 3.2: A linha tracejada marca a fronteira entre a CPU e a GPU nos estágios do processamento gráfico.

A GPU processa cada primitiva com todas as informações que a constituem, como vértices e texturas. As GPUs modernas podem ser programadas de duas formas: programas de vértices, que atuam sobre dados relativos a vértices e programas de fragmentos, que atuam sobre os dados relativos a fragmentos. Quando a GPU finaliza seu trabalho, um quadro está completamente desenhado. Cada chamada de uma função gráfica da API, realizada pela aplicação, necessariamente passará por todos os estágios da figura 3.1.

A relação entre os diversos estágios é complexa, pois alguns estágios são executados por processadores diferentes – figura 3.2. O fato de mais de um processador estar envolvido introduz um acoplamento entre estes processadores. A medida de relógio de clock não é a mesma para os processadores e existe largura de banda finita para a conexão entre os processadores.

```
BeginScene();
      ...
      SetTexture(...);
      DrawPrimitive(D3DPT_TRIANGLELIST, 0 ,1);
      ...
EndScene();
Present();
```

Figura 3.3: Exemplo de comandos de desenho disponibilizados pela API DirectX.

O pipeline de processamento DirectX possui um "buffer" que armazena os comandos antes de serem enviados para o driver. Esse buffer é responsável por armazenar instruções de desenho até que a aplicação pare de enviar instruções ou até que o buffer esteja cheio. Quando um desses dois eventos ocorre, temos uma transição de estado – figura 3.4. Uma transição de estado acontece quando o controle da CPU passa da API para o driver e a CPU passa a operar com privilégios que pertencem apenas ao Sistema Operacional. Quando o driver finaliza as tarefas, uma nova transição ocorre; saindo do modo privilegiado indo para o modo de usuário. O buffer de

comandos da API não tem tamanho rígido, podendo mudar durante o tempo ou de uma versão para a seguinte da API. Existem outros eventos que podem levar a uma transição de estado, tais como a chamada da função `IDirect3DQuery9::GetData` com argumento `D3DGETDATA_FLUSH`, que será utilizada mais adiante.

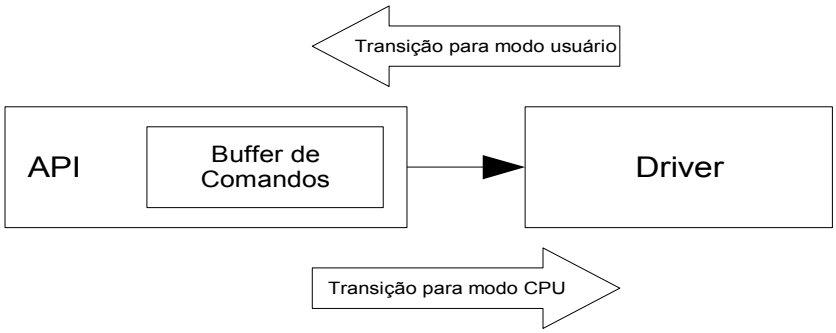

Figura 3.4: O Buffer de Comandos acumula comandos para enviar em lotes; a cada envio de instruções para o driver ocorre uma transição de estado.

O pacote de desenvolvimento para DirectX disponibiliza um conjunto de ferramentas para medição de performance de aplicativos que utilizam essa API. Alguns fabricantes de hardware gráfico também disponibilizam algumas ferramentas para medição de performance nos seus conjuntos de desenvolvimento. Algumas dessas ferramentas interagem diretamente com o driver da GPU, disponibilizando para a aplicação acesso a alguns contadores implementados em posições estratégicas do hardware, como é o caso do NVPerfHUD (REGE, 2004).

Algumas dessas aplicações não são adequadas para utilização no presente trabalho. O PIX, ferramenta de avaliação de performance (DIRECTX, 2004) distribuída junto com o conjunto de desenvolvimento do DirectX, não permite salvar os dados coletados em um arquivo com formato que não seja o seu próprio, que é proprietário. O NVPerfHUD da Nvidia não salva os dados coletados para arquivos, apenas apresenta-os na tela, sobrepostos à aplicação. Outro método para a medição de performance foi escolhido devido a limitação dessas aplicações. A estratégia escolhida foi a instrumentação do código da aplicação, como será discutido na próxima seção.

3.2 Estratégia Adotada para Medição de Performance

O processamento gráfico é realizado em vários estágios e nem todos estes estágios rodam no mesmo processador – figura 3.2. É importante saber de qual parte do pipeline se está realmente medindo a performance. A aplicação, em geral, não tem acesso ao driver e apenas informa o código que será executado na GPU. Como a GPU não disponibiliza mecanismos de medição de performance é necessário a utilização de mecanismos de medida disponibilizados pelo sistema operacional e pela API – ambos rodando na CPU. A estratégia adotada é a instrumentação do código da aplicação que executa na CPU para, com auxilio de funções implementadas pela API, medir o tempo que a GPU leva para executar seu trabalho. Será usado um contador de alta performance disponibilizado pelo sistema operacional.

Alguns sistemas incluem um contador de performance de alta resolução que provê tempos decorridos com grande precisão. Este contador é acionado pelo relógio interno da própria CPU e a cada número definido de ciclos de relógio tem o seu valor incrementado. Se esse contador existe no sistema, a função `QueryPerformanceFrequency` disponibilizada pelo sistema operacional MS Windows XP pode ser utilizada para expressar a freqüência deste contador em contagens por segundo. O valor dessa contagem é dependente do processador; para cada freqüência de operação da CPU um valor diferente é utilizado. Existe outra função que retorna a leitura atual deste contador de alta performance e que é utilizada para inferir intervalos de tempo.

As medidas realizadas neste trabalho foram feitas através de instrumentação do código da aplicação – figura 3.5 – utilizando o contador de alta precisão do Sistema Operacional. A função `QueryPerformanceCounter (LARGE_INTEGER *lpPerformanceCount)` lê o valor do contador de alta performance e o armazena na variável *`lpPerformanceCount`*. A estratégia para medir o tempo gasto para execução de uma parte do código é registrar o valor do contador antes e depois de executar a seção de código. Assumindo, por exemplo, que `QueryPerformanceFrequency` indica que a freqüência do contador de alta performance é 50.000 contagens por segundo. Se uma aplicação chama `QueryPerformanceFrequency` imediatamente antes e imediatamente depois de uma seção de código, os valores retornados por `QueryPerformanceCounter` podem ser 1500 e 3500, respectivamente. Estes valores indicariam um tempo transcorrido de 0,04 segundos (2000 contagens) enquanto o código é executado. Este cálculo pode ser expresso como:

$$\text{Tempo} = \frac{\textit{contagem final} - \textit{contagem inicial}}{\textit{contagens por segundo}} = \frac{3500 - 1500}{50000} = 0{,}04\,s \quad (3.3)$$

A instrumentação apresentada acima não leva em consideração as transições de estado que podem ocorrer entre o envio de um comando da API e o seguinte (entre `SetTexture` e `DrawPrimitive`, na figura 3.5, por exemplo). Para contornar este problema, é utilizado um mecanismo que garante que o buffer de comando esteja vazio

```
BeginScene();
      ...
      QueryPerformanceCounter(&start);

      SetTexture(...);
      DrawPrimitive(D3DPT_TRIANGLELIST, 0 ,1);

      QueryPerformanceCounter(&stop);
      ...
EndScene();
Present();
```

Figura 3.5: Exemplo de código da aplicação instrumentado para utilização do contador de alta performance.

quando a contagem de tempo é iniciada. Existem dois comandos que auxiliam na tarefa de saber quando não existem mais comandos de desenho a serem despachados para o pipeline gráfico. Uma chamada de função `Idirect3DQuery9::Issue` com o argumento `D3DISSUE_END` coloca um marcador na fila de comandos de primitivas. Este marcador

indica para o driver que todos os comandos que o precedem devem ser despachados e executados pela GPU. Este comando faz com que novos comandos de desenho sejam enviados para a GPU somente depois dela ter finalizado os comandos que o precediam. Finalmente uma chamada ao comando `Idirect3DQuery9::GetData` com o argumento `D3DGETDATA_FLUSH` somente retorna verdadeiro quando todas as instruções do buffer de comando foram enviadas para o driver e para a GPU. O procedimento para medir o tempo gasto pela instruções de desenho da API é descrito a seguir:

- Adicionar um marcador na fila de comandos que serão enviados para a GPU. Isto é feito chamando `Idirect3DQuery9::Issue` com o argumento `D3DISSUE_END`. Este marcador indica ao driver para controlar quando todos os comandos que o precedem terminaram de executar e a GPU está ociosa – figura 3.6.

- Esperar o buffer de comandos ficar vazio invocando `Idirect3DQuery9::GetData` com o argumento `D3DGETDATA_FLUSH`. Este comando garante que as instruções remanescentes no buffer de comandos sejam definitivamente enviadas para o driver e finalmente para a GPU. Este comando não retorna verdadeiro enquanto ainda houver algum comando no buffer que precise ser enviado para o driver e para GPU. Com um laço, fica-se verificando constantemente se o comando de esvaziamento do buffer já retornou. O processamento somente prossegue quando o programa sai deste laço.

- Anotar o valor inicial do contador de alta performance mediante uma chamada a `QueryPerformanceCounter`.

- Enviar o(s) comando(s) que dispara(m) o desenho do quadro

- Adicionar um novo marcador na fila de comandos enviados para a GPU.

- Esvaziar o buffer de comandos novamente, de maneira similar a utilizada anteriormente e espera-se até que a GPU tenha finalizado seu processamento.

- Anotar o valor do contador de alta performance novamente.

- Obter a diferença entre o valor inicial e o valor final do contador de alta performance.

A diferença entre as duas medidas do contador, dividido pelo número de contagens que ele executa a cada segundo, indica o tempo necessário para a execução dos comandos contidos entre as duas leituras do contador. Se foram enviados comandos necessários para desenhar apenas um quadro, então este é o tempo requerido para o processamento de apenas um quadro pela GPU. O inverso desse número é a quantidade instantânea de quadros por segundo (FPS) que a GPU está desenhando. Essa medida é dita instantânea, porque indica o número de quadros que estão sendo desenhados em um determinado instante de tempo. Esse número pode variar com o tempo, fruto da carga a qual a GPU pode estar sendo submetida.

O número de quadros desenhados pode ser muito alto sob certas condições, ultrapassando até mesmo mais de quatro mil quadros desenhados em um único segundo. Nessas condições, outras flutuações de carga que ocorrem naturalmente no hardware –

como outros processos requisitando a atenção da CPU e da GPU, ou algum periférico enviando uma interrupção – podem alterar muito a performance instantânea medida. Para evitar as flutuações de mais alta freqüência, optou-se por realizar apenas uma medida por segundo, em vez de anotar o tempo gasto por cada um dos quadros. Os valores medidos são então enviados para um arquivo texto puro no disco rígido do computador, o que também justifica esta política de amostragem, evitando gargalos em outros subsistemas do computador utilizado. O valor final é então obtido como a média aritmética de uma série de valores armazenados.

```
BeginScene();
      ...
      pEvent->Issue(D3DISSUE_END);
      while(S_FLASE==pEvent->GetData(NULL,0,D3DGETDATA_FLUSH));

      QueryPerformanceCounter(&start);

            SetTexture(...);
            DrawPrimitive(D3DPT_TRIANGLELIST, 0 ,1);

      pEvent->Issue(D3DISSUE_END);
      while(S_FLASE==pEvent->GetData(NULL,0,D3DGETDATA_FLUSH));

      QueryPerformanceCounter(&stop);
      ...
EndScene();
Present();
```

Figura 3.6: Exemplo de código da aplicação instrumentado e com comandos para o esvaziamento do buffer de comandos da API.

A instrumentação direta do código a ser executado na GPU não é possível pois não existem funções de medida de performance semelhantes a estas disponíveis no hardware gráfico. Apenas o driver do hardware tem capacidade de verificar quando a GPU terminou o seu processamento e está ociosa.

É importante observar que o driver pode forçar com que o subsistema gráfico opere sincronizado com o dispositivo de display, o que efetivamente diminui a performance de toda a aplicação. Quando este modo de operação está habilitado, dizemos que o sincronismo de vídeo está ligado. Para as medidas realizadas neste trabalho o sincronismo de vídeo esteve sempre desligado, permitindo que a GPU trabalhe independentemente da velocidade de exibição do display gráfico.

3.3 Precisão das Medidas

Na análise de performance de sistemas computacionais é importante ter em conta quando um intervalo de tempo é significativo para as medidas realizadas. Neste trabalho a CPU está sendo utilizada para medir tempos relativos a trabalhos realizados na GPU. Deve-se levar em consideração se a CPU está apta a medir com confiança os intervalos de tempos de trabalho típicos da GPU. Por exemplo, em uma CPU que opera em freqüências entre 2 e 3 Ghz, o contador de alta performance faz cerca de 3 milhões de contagens por segundo. A precisão do contador de alta performance é da ordem de centenas de micro-segundos, ou 10^{-4} segundos.

Um processador de fragmentos típico trabalha em freqüências superiores a 350 Mhz. Se a GPU está equipada com 16 pipelines, por exemplo, este processador consegue operar sobre cerca de $5{,}6 \cdot 10^9$ fragmentos a cada segundo (cada pipeline executa um ciclo do programa de fragmentos em um fragmento a cada ciclo de relógio). Um quadro que ocupe toda a tela do computador numa resolução de 1024x768 pixels, tem menos de um milhão de fragmentos. Isso resulta num pico de processamento que supera os 5 mil quadros por segundo.

$$\text{FPS} = \frac{frags/s}{frags/quadro} = \frac{5{,}6 \times 10^9}{1024 \times 768} \simeq 5000\, FPS \tag{3.4}$$

Segundo a equação 3.1, se são desenhados cerca de 5 mil quadros por segundo, isso significa que cada quadro gasta cerca de $2 \cdot 10^{-4}$ segundos para ser desenhado. Se o número de eventos do contador de alta performance transcorridos durante o tempo de desenho de um quadro é muito baixo, mesmo pequenas oscilações nas medidas de tempo podem desviar muito os resultados obtidos. Sendo o número de eventos grande frente a resolução do contador, é garantido que pode ser obtida uma boa precisão nos números quando medindo o tempo gasto para o desenho de cada quadro. Amostrando estes valores em intervalos regulares – uma vez por segundo nas medidas atuais – também garante a independência das oscilações de carga de mais alta freqüência. Utilizar a média de diversas amostras, auxilia ainda mais na remoção de oscilações (ou ruído).

3.4 Características de Performance

Alguns parâmetros do sistema devem ser variados para que seja possível caracterizar como a performance de um sistema se comporta com a sua variação. Agora que se sabe como realizar as amostragens para as medidas de performance, deve-se decidir quais serão estes parâmetros a variar. Os primeiros parâmetros escolhidos para descrever o comportamento da performance do hardware a ser estudado foram os que descrevem a performance máxima que um processador de fragmentos pode atingir:

- A freqüência de operação do processador de fragmentos. Como cada pipeline pode aplicar um ciclo do programa em um fragmento a cada ciclo do relógio, este número dá a performance máxima obtida por pipeline.

- O número de pipelines presente em cada processador de fragmentos. Indica quantos fragmentos podem ser processados simultaneamente pelo processador de fragmentos.

- O número de ciclos necessários para a execução completa do programa de fragmentos. Este parâmetro é um indicativo da carga de trabalho que deverá ser realizado sobre cada dado a ser processado.

- O número de dados a serem processados. O foco deste estudo são os fragmentos (dados) que compõem um quadro.

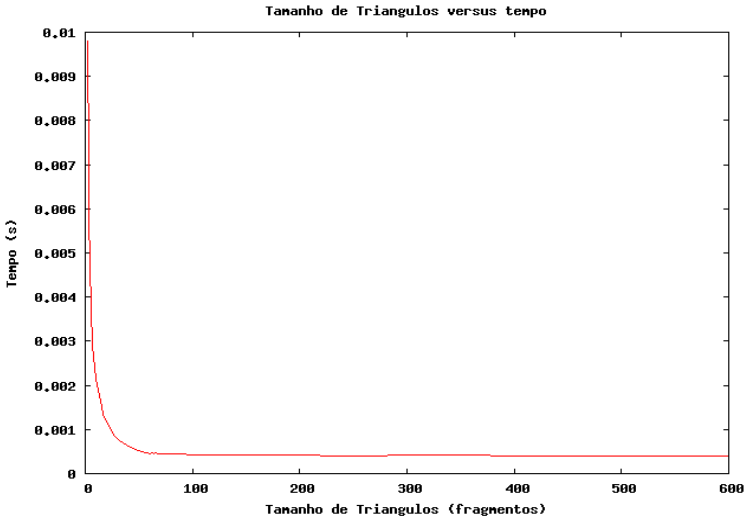

Figura 3.7: Performance como função do tamanho de triângulos

Além dos parâmetros que definem a performance máxima de um processador de fragmentos é interessante analisar parâmetros que podem ser responsáveis por redução de performance do processador de fragmentos. Os parâmetros escolhidos para serem abordados neste trabalho são:

- O tamanho dos triângulos desenhados em cada quadro
- O tamanho do bloco de coerência na execução de desvios de código dos fragmentos sendo processados
- O tamanho das texturas sendo acessadas
- O tipo de texturas – isto é, bytes por texel

Alguns destes recursos têm seus valores fixos no modelo de hardware utilizado. O número de pipelines, por exemplo, não pode ser alterado, para medições em uma mesma placa gráfica. Mesmo parâmetros que não podem ser variados devem ser levados em conta porque descrevem características fundamentais do hardware. Tomando novamente como exemplo o número de pipelines de fragmentos, este indica um grau de paralelismo do hardware sendo analisado.

Dentre os recursos que podem ser variados, medidas preliminares indicam que a variação de alguns destes parâmetros têm impacto maior sobre a performance do que outros, como por exemplo o tamanho dos triângulos – figura 3.7. Outros recursos apresentam comportamento bastante linear, como por exemplo a variação do número de fragmentos – figura 3.8.

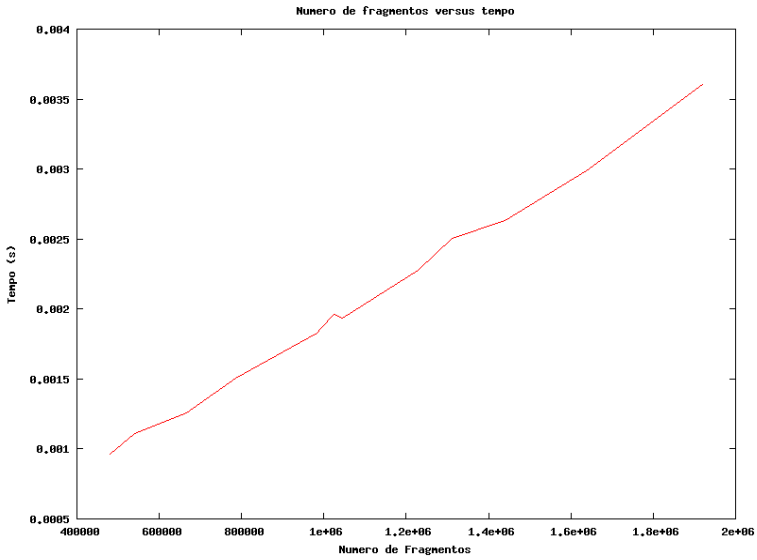

Figura 3.8: Variação da performance com o número de fragmentos processados

Estes recursos foram escolhidos por se julgar os mais representativos de possíveis vínculos entre os diversos componentes do processador de fragmentos. Estes recursos também são os mais utilizados pela maioria das avaliações de performance publicadas e também são os recursos que o programador tem acesso e maior grau de controle em aplicações.

4 Modelo de Performance

Um modelo pode ser visto como uma construção teórica que representa um processo (JACOBY, 1980). Modelos são utilizados primariamente para descoberta de novos fatos, prover argumentos lógicos para explicações, avaliação de hipóteses teóricas e construção de procedimentos experimentais para teste dessas hipóteses. As deduções de um modelo são ditas idealizadas porque o modelo faz suposições que podem não ser verdadeiras em algum nível de detalhamento, mas permitem a produção de resultados que são precisos dentro de um limite de erro. Em alguns casos modelos podem ser utilizados para implementar simulações computacionais que ilustram o comportamento do sistema.

Um modelo matemático descreve um sistema por um conjunto de variáveis e um conjunto de equações que estabelecem relações (vínculos) entre as variáveis (DYM, 1980). As variáveis descrevem propriedades do sistema, como por exemplo contadores, medição de tempo, ocorrência de eventos, etc. O modelo de fato é o conjunto de funções que descreve a relação entre as diferentes variáveis. Pode-se destacar 5 grupos básicos de variáveis usadas na prática:

- **decisão ou independentes**: representam as propriedades que serão manipuladas para avaliação do comportamento do modelo.
- **estado**: são variáveis que selecionam um dentre diversos estados possíveis para uma dada propriedade do sistema.
- **constantes**: representam propriedades fixas do sistema modelado, que não são alteradas pelo modelo.
- **variáveis aleatórias**: utilizadas em modelos que admitem propriedades não determinísticas.
- **saída**: representam o resultado das operações de vínculo entre as demais variáveis, e representam a saída do modelo.

Os modelos matemáticos possuem diferentes propriedades que permitem classificá-los de diversas formas:

- **linear** vs. **não-linear**: O modelo é dito linear se as funções e vínculos são representados inteiramente por equações lineares. Se uma ou mais das funções ou vínculos é representado por uma equação não linear o modelo é dito não linear.

- **determinístico** vs. **probabilístico**: Um modelo é dito determinístico quando a probabilidade de ocorrência de qualquer evento que ele esteja modelando é sempre 1 (ou 100%). Em um modelo probabilístico (ou estocástico), a probabilidade da ocorrência de um ou mais eventos modelados pode não ser 100%.

- **estático** vs. **dinâmico**: Um modelo estático não tem dentre as suas variáveis (exceto as de saída) o elemento tempo. Um modelo dinâmico considera o tempo como uma de suas variáveis utilizadas.

Problemas de modelagem matemática seguidamente são classificados como "caixa-preta" ou "caixa-transparente", de acordo com quanta informação do sistema está disponível a priori (BRATLEY, 1987). Um modelo "caixa-preta" representa um sistema do qual não se tem praticamente nenhuma informação a priori. Um modelo "caixa-transparente" é um sistema no qual toda informação necessária está à disposição. Praticamente todos os sistemas estão em algum ponto entre os dois extremos, portanto este conceito serve como um guia intuitivo. Usualmente é preferível utilizar tanta informação quanto possível para tornar o modelo mais preciso. Modelos "caixa-transparente" são considerados mais precisos, porque se informações corretas forem utilizadas o modelo irá se comportar mais próximo da realidade. Informação a priori, seguidamente, vem na forma de conhecimento sobre o tipo de funções que relacionam as diferentes variáveis. Em modelos "caixa-preta" alguém pode tentar estimar tanto a forma dos relacionamentos entre as variáveis quanto os parâmetros numéricos nestas funções. Sem informação a priori deve-se tentar utilizar funções tão gerais quanto possível, para tentar cobrir todos os diferentes modelos que podem estar sendo representados. O problema de utilizar um grande conjunto de funções para descrever um sistema é que estimar os parâmetros se torna uma tarefa muito difícil quando a quantidade de parâmetros cresce. Qualquer modelo que não é totalmente uma "caixa-transparente" contém alguns parâmetros que são utilizados para ajustar o modelo ao sistema que ele irá descrever. Se a modelagem é feita por uma rede neural, por exemplo, a otimização dos parâmetros é chamada de treinamento. Em modelagem mais convencional, através de funções matemáticas, os parâmetros são determinados por um ajuste de curvas.

A complexidade de um modelo é um fator importante a ser levado em conta no seu desenvolvimento. Quanto maior o nível de detalhamento requerido pelo modelo, maior o número de funções requeridas para a sua descrição. Quanto maior o número de equações envolvidas, maior será o custo computacional para a resolução destas equações. Num caso extremo a simulação pode ser computacionalmente mais cara do que a execução (ou construção) do sistema modelado. A incerteza também poderia aumentar, pois para cada equação adicionada, mais parâmetros são necessários. Um pouco de incerteza sempre é adicionada com cada parâmetro. O modelo de gravitação proposto por Newton, por exemplo, é uma aproximação que não leva em conta fatores como a dilatação do espaço-tempo para altas velocidades. Mas esse modelo serve muito bem para descrever a gravitação de corpos celestes quando as dimensões não são microscópicas nem as velocidades próximas à da luz.

Uma parte importante do processo de modelagem é a avaliação do modelo proposto. Como saber ser um modelo matemático descreve bem o sistema? Esta não é uma questão fácil de responder. Usualmente tem-se um conjunto de medidas do sistema que são utilizadas na criação do modelo. Se o modelo foi bem construído, o modelo irá

adequadamente mostrar as relações entre as variáveis do sistema para estas medidas. Mas aqui surgem algumas questões: Como saber se o modelo descreve bem as propriedades do sistema entre os dados medidos (interpolação)? Como saber se o modelo descreve bem eventos fora da faixa de dados medidos (extrapolação)? A abordagem mais comum é dividir os dados medidos em duas partes: dados de ajuste e dados de verificação. Os dados de ajuste são utilizados para ajustar o modelo, isto é, estimar parâmetros utilizados pelo modelo. Os dados de verificação são utilizados para avaliar o modelo.

Este capítulo propõe um modelo que toma como entrada parâmetros que descrevem o hardware gráfico e a aplicação a ser executada, e gera como saída uma estimativa de performance. O foco do modelo é o processador de fragmentos da GPU, sendo assumido que os demais estágios do pipeline gráfico têm uma carga muito menor do que a do processador de fragmentos – ou seja, os demais estágios não serão limitadores da performance (gargalos) da execução do programa. O modelo permite estimar a performance do hardware em relação aos recursos apresentados no capítulo anterior. O modelo proposto é composto por módulos associados, abstratamente em série e/ou em paralelo. A decomposição modular visa controlar o nível de complexidade a ser representada por cada módulo (ou parte) do modelo. A funcionalidade de cada módulo deve ser, idealmente, independente da funcionalidade de todos os outros módulos do modelo. A resposta do modelo é dada pela composição das respostas dos módulos.

4.1 Características do Modelo

A proposta de um modelo da performance da GPU é oferecer uma aproximação do comportamento de performance de uma aplicação gráfica. Aplicações que utilizam o processamento em GPUs, tais como programas de ray-tracing, simulação de fluídos, simulação de movimento de muitos corpos, poderiam ter um ganho de performance se fossem efetuadas modificações na maneira em que são implementadas ou mapeadas para a arquitetura da GPU. O modelo deve satisfazer algumas características:

- **linearidade:** a utilização de equações lineares reduz o conjunto de relacionamentos possíveis entre as variáveis. Com um número reduzido de relacionamentos entre as variáveis o controle sobre estes relacionamentos torna-se mais simples.

- **determinismo:** os resultados apresentados pelo modelo, devem ser sempre reproduzíveis: para os mesmos dados de entrada, saídas iguais devem ser geradas. O hardware gráfico é determinístico: para um mesmo conjunto de dados e instruções, o resultado produzido pelo hardware gráfico é o mesmo.

- **invariância temporal:** o tempo não é considerado como uma variável de decisão, nem como parâmetro ou variável de estado. O tempo é representado apenas como uma variável de saída.

Existem outras características que são desejáveis na construção do modelo:

- **modularidade** – a implementação em estágios torna o modelo mais simples e controlável, uma vez que os resultados de uma etapa servem como

entrada para a próxima. É desejável que os estágios possam ser combinados de mais de uma maneira distinta. A combinação dos estágios permite ao simulador imitar o comportamento das diferentes associações existentes entre os recursos do hardware.

- **modelo de desaceleração** – o modelo deve prever uma performance máxima possível e então com base nos parâmetros de entrada, essa performance é reduzida até à obtenção da estimativa de performance final. Na GPU, cada recurso que é utilizado diminui a performance de acordo com a carga a que está sujeito.

- **o modelo deve produzir resultados em unidades comparáveis às medidas práticas** – esta característica é importante para avaliação do modelo. Os números retornados pelo modelo devem ser em unidades relevantes para a quantificação sendo realizada, por exemplo Quadros Por Segundo (FPS, do inglês *Frames Per Second*) ou tempo necessário para o desenho de um quadro.

- **o modelo deve ser alimentado com informação sobre a utilização dos recursos** – o modelo deve considerar recursos disponibilizados pelo hardware, tais como freqüência de operação do pipeline, número de pipelines, número de fragmentos a serem processados, número de instruções presentes nas classes de programas que se pretende utilizar no hardware gráfico e características da aplicação – acessos a texturas, tamanho de triângulos e coerência dos blocos de desvios.

4.2 Proposição do Modelo

O modelo proposto segue a caracterização discutida na seção anterior. Em termos de linearidade, o modelo utiliza funções que são lineares por partes em cada um dos módulos (estágios). Em termos de determinismo o modelo garante que quando alimentado com dados iguais, a saída será sempre a mesma. Em termos de invariância temporal, o modelo não utiliza o tempo como um de seus dados de entrada. O modelo estima a performance da GPU, quando alimentado com parâmetros da GPU, do programa e dos dados a processar. O modelo é composto de partes que podem ser associadas, abstratamente, em paralelo ou em série.

O modelo usa como parâmetros de entrada os mesmos recursos que se escolheu como relevantes para estimativa de performance – seção 3.4:

- A freqüência de operação do processador de fragmentos
- O número de pipelines presente em cada processador de fragmentos
- O número de ciclos do programa sendo executado no processador de fragmentos
- O número de fragmentos por quadro
- O tamanho dos triângulos desenhados em cada quadro
- O tamanho do bloco de coerência dos fragmentos sendo processados
- O tamanho das texturas sendo acessadas

- O formato de armazenamento das texturas

O modelo proposto é composto de vários estágios: um estágio inicial e uma série de módulos adicionais que determinam penalidades baseadas nas características da aplicação – figura 4.1. O estágio inicial fornece os resultados que seriam obtidos com a performance máxima. Os demais módulos são responsáveis por estimar a perda de performance devido a carga imposta aos recursos oferecidos pela GPU. Estes módulos são alimentados com uma série de parâmetros que são informados ao modelo pelo usuário. A seguir serão descritas as partes do modelo e como elas estão associadas umas com as outras.

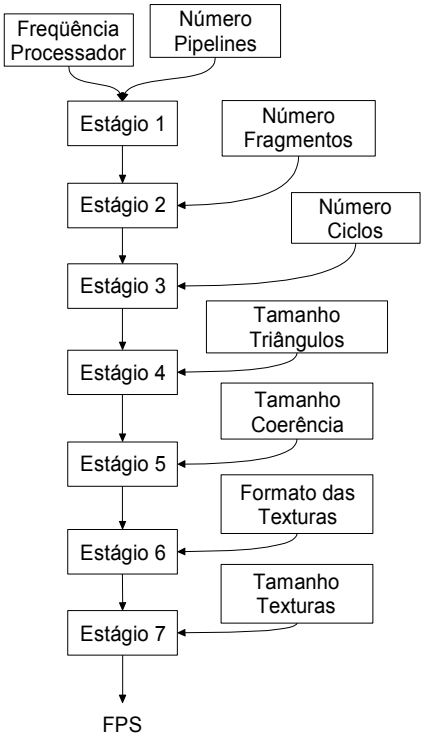

Figura 4.1: Estrutura do modelo Proposto; O modelo é composto de estágios (lógicos) que vão aproximando sucessivamente a estimativa de performance.

4.2.1 Modelo Básico

Os dois primeiros estágios do modelo – figura 4.1 – são responsáveis pela estimativa de performance máxima para uma aplicação. O regime de mais alta performance da GPU requer que o programa de fragmentos utilizados não contenha instruções de tomada de desvio dinâmicas (sempre toma ou nunca toma os desvios), utilize triângulos grandes (mais de 1000 fragmentos por triângulo) e não realize acessos a texturas. Os parâmetros de entrada que permitem a estimativa de performance máxima são:

- **velocidade do processador** – velocidade do processador de fragmentos, freqüência de operação, medida em ciclos por segundo (Hz). O processador de fragmentos tem a capacidade de aplicar um ciclo do programa de

fragmentos em um fragmento que está em um pipeline a cada ciclo da freqüência de operação.

- **número de pipelines** – o processador de fragmentos em geral possui mais de um pipeline, sendo que por cada pipeline processa um ciclo do programa por fragmento por ciclo de relógio.

Os próximos estágios acrescentam fatores de redução da performance máxima. A performance diminui quando os parâmetros dos dois próximos estágios são diferente de 1. Quando estes fatores aumentam, a performance diminui, sugerindo que existe uma relação inversamente proporcional entre eles e a performance:

- **número de ciclos** – número de grupos de instruções que compõem um programa. Uma VLIW é um agrupamento de instruções que é despachado pelo pipeline e vai reprogramando as unidades funcionais deste pipeline (PATTERSON, 1996). Os programas de fragmentos são descritos pelo número de vezes (ciclos) que um fragmento vai ter que passar pelo pipeline.

- **número de fragmentos** – é o número de fragmentos que passarão pelo processador de fragmentos e sobre os quais será aplicado o programa de fragmentos em questão. Depende da aplicação que gera os fragmentos.

O modelo básico é representado como uma função destes parâmetros, que são mantidos constantes durante toda execução do programa de fragmentos:

$$F_{base}(\mathit{freqüência}, \mathit{pipes}, \mathit{ciclos}, \mathit{fragmentos}) \tag{4.1}$$

Estas informações estão relacionadas aos dados que serão processados e também à arquitetura que o modelo estará aproximando. Por exemplo, o número de pipelines indica o grau de paralelismo de um modelo de hardware específico. O número de ciclos gasto por um determinado programa também informa sobre a eficiência da implementação dessa função no hardware.

A performance do hardware gráfico em uma situação como a apresentada aqui (código sem desvios, triângulos grandes, grande número de fragmentos e sem acesso a texturas) deve ser diretamente proporcional tanto à freqüência de operação do processador de fragmentos quanto ao número de pipelines presentes na GPU. Quanto mais pipelines, mais fragmentos podem ser processados simultaneamente. Aumentando a velocidade de operação do processador de fragmentos mais fragmentos podem ser processados num mesmo intervalo de tempo. A performance também deve ser inversamente proporcional ao número de instruções presentes no programa sendo executado e ao número de fragmentos que deverão ser processados. Quando o número de instruções num programa é aumentado, mais ciclos serão necessários para execução completa do programa sobre um dado conjunto de fragmentos. Quanto maior o número de fragmentos a serem processados, maior será o número de vezes que um pipeline ficará cheio e portanto maior o número de vezes que um programa deverá ser executado. Dadas as relações de proporcionalidade de cada um dos parâmetros de entrada descritos, chega-se à seguinte equação:

$$F_{base}(\mathit{freqüência}, \mathit{pipes}, \mathit{ciclos}, \mathit{fragmentos}) = \frac{\mathit{freqüência} * \mathit{pipes}}{\mathit{ciclos} * \mathit{fragmentos}} \quad (4.2)$$

sendo o valor de F_{base} dado em Quadros por Segundo (FPS), *freqüência* é a freqüência de operação do processador de fragmentos, a sua velocidade, dado em Hz; *pipes* é o número de pipelines de fragmentos presentes na GPU; *ciclos* o número de ciclos de instruções do programa de fragmentos sendo executado e *fragmentos* o número de fragmentos que necessitam passar pelo processador e executar o programa.

Programas de fragmentos sem desvios de fluxo de código são bem descritos pela relação apresentada acima. Para programas de fragmentos que utilizam desvios de código, a formulação apresentada precisa ser refinada, conforme visto na próxima seção.

4.2.2 Desvios de fluxo

Quando instruções de desvio estão incluídas num programa de fragmentos, estas instruções alteram o fluxo de execução do programa. Portanto, o número de ciclos para execução do programa deixa de ser apenas uma contagem seqüencial. Alterações de fluxo de execução de um programa tem o perigo potencial de introduzir bolhas no pipeline de execução de instruções. Além do custo da própria instrução de desvio esse tipo de instrução acarreta uma sobre-carga no processamento devido à necessidade de avaliação da tomada ou não do desvio.

Como visto no capítulo 2, quando instruções estão incluídas dentro de estruturas de controle de fluxo, essa sobrecarga é de dois ciclos para cada instrução de controle de fluxo aplicada a cada fragmento nas GPUs das famílias 6 e 7 da NVidia. Isso resulta em blocos de instruções cercados por instruções de controle de fluxo. As instruções de controle de fluxo adicionam dois ciclos no início e outros dois no final da execução de cada bloco de instruções. Blocos de código são contidos por duas instruções de desvio, de forma que ao número de ciclos de um dado bloco, mais 4 ciclos devem ser somados (dois por cada instrução de desvio).

Assumindo que todos os blocos de código contidos em todos os desvios são executados (ou seja, o pior caso para código com desvios), a fórmula para F_{base} poderia ser expressa como:

$$F_{base}(\mathit{freqüência}, \mathit{pipes}, \mathit{ciclos}, \mathit{frags}) = \frac{\mathit{freqüência} * \mathit{pipes}}{(\mathit{ciclos} + 4 * n + \sum \mathit{ciclos}_n) * \mathit{fragmentos}} \quad (4.3)$$

onde ciclos é o número de ciclos fora dos blocos de desvio, n é o número de blocos e ciclos_n o número de ciclos contidos dentro destes blocos. Como será visto mais adiante, nem sempre todos os blocos contidos por instruções de desvio são executados, o que acarreta em ganho de performance.

4.2.3 Tamanho de triângulos

Dados de entrada constantes – como os utilizados para a F_{base} – nem sempre descrevem suficientemente bem uma aplicação – isto é, pode haver dependência de

performance em parâmetros não considerados na F_{base} : por exemplo, acessos a texturas e triângulos de um tamanho abaixo do que é assumido em F_{base} . A variação dos dados que estão sendo processados, tais como o número de triângulos também deve influenciar o comportamento do modelo. Estes dados devem ser enviados ao simulador, juntamente com os parâmetros anteriores. O modelo passa a ser descrito como uma combinação de funções:

$$\mathrm{FPS} = f(\mathrm{F}_{base}, \mathrm{G}(\text{tamanho\_triângulo})) \tag{4.4}$$

onde $f()$ é uma função que depende da estimativa de performance anterior e de um fator que vai introduzir uma perda de performance em função do tamanho dos triângulos (em fragmentos) sendo processados. A equação (4.4) recebe como entrada a estimativa de performance do estágio anterior F_{base} juntamente com novos dados de entrada para produzir uma nova aproximação da estimativa de performance. O modelo agora precisa carregar informações sobre os triângulos sendo processados. Para isso, será empregado um modelo de desaceleração dependente do tamanho dos triângulos.

A variação da performance com o tamanho do triângulo é aproximada como sendo linear por partes. Assim, o comportamento deste recurso é aproximado por um conjunto de funções lineares, cujos parâmetros são determinados a partir da execução de uma série de pequenos programas de fragmentos e de CPU. Um número maior de retas permite aproximar melhor os dados modelados dos dados medidos. A função que relaciona a performance ao tamanho dos triângulos pode ser descrita como:

$$\mathrm{G}_{triang} = \begin{cases} a_1 + b_1 * x, & x_0 < x \le x_1 \\ a_2 + b_2 * x, & x_1 < x \le x_2 \\ \vdots & \\ a_n + b_n * x, & x_{n-1} < x \le x_n \end{cases} \tag{4.5}$$

onde a_n e b_n são os coeficientes das retas que serão ajustadas sobre os dados medidos para uma região entre x_{n-1} e x_n, e x é o tamanho do triângulo em fragmentos.

Assim como o tamanho dos triângulos utilizados afeta a performance, a freqüência com que desvios são tomados (o que está relacionado com o tamanho do bloco de coerência entre os fragmentos que vão ou não tomar um desvio), também acarreta em redução de performance, o que é discutido a seguir.

4.2.4 Blocos de coerência

O comportamento da GPU muda conforme a freqüência com que desvios de fluxo de execução são tomados. Esta freqüência da tomada de desvios é reflexo da coerência dos dados que decidem a tomada do desvio: quanto mais coerentes, isto é, mais uniforme forem os dados, tão mais uniformemente os desvios serão tomados (ou deixarão de ser tomados). Os dados podem se apresentar em grupos onde todos são coerentes. Isto é, todos os dados a serem processados dentro de um grupo podem decidir pela tomada de um desvio. Quanto menor forem estes blocos de coerência, pior será a

performance, pois blocos menores significam tomadas de desvio mais freqüentemente. Tomadas de desvio podem introduzir bolhas no pipeline (PATTERSON, 1996) e portanto degradar a performance do pipeline.

A performance como função do tamanho de triângulos não é completamente independente da performance como função da freqüência com que desvios são tomados. Uma vez conhecido o comportamento de cada um dos parâmetros, será analisado o seu comportamento em conjunto. Ou seja, será produzido um gráfico de três dimensões que relaciona a função entre o tamanho dos triângulos e o tamanho dos blocos de coerência. Caso não seja possível encontrar uma composição linear de funções que possa descrever corretamente a relação entre as funções de triângulos e as funções de bloco de coerência, várias funções lineares – equação (4.5) – serão determinadas para diversos tamanhos de bloco de coerência. A partir desta família de funções, os demais valores são interpolados. Para cada tamanho de bloco de coerência, uma $G_{triângulo}$ deve ser utilizada resultando em uma nova $G_{triangulos,coerência}$.

A próxima seção adiciona mais um fator de desaceleração ao modelo, para considerar acessos a texturas. Será descrito como as características da textura e do acesso a memória afetam a performance.

4.2.5 Acesso a texturas

Acessos a texturas afetam a performance das GPUs, juntamente com os parâmetros discutidos nas seções anteriores. Acessos a texturas correspondem a acessos a memória, quando é requisitada a leitura de um valor durante a execução de um programa de fragmentos. Se o dado não está disponível, o processamento do fragmento é parado até que este dado retorne a partir da memória. A ligação que existe entre o processador de fragmentos e os bancos de memória tem uma taxa de leitura/escrita que é finita. Esta taxa de acessos que uma memória consegue sustentar é governada pela velocidade dos acessos e pela quantidade de dados que podem ser entregues a cada operação. Chamamos esta taxa de "*largura de banda*" que pode ser medida em unidades de quantidade de dados (bytes) por unidade de tempo (segundos). A largura de banda que existe entre os pipelines do processador de fragmentos e a memória impõem um limite máximo na velocidade com que dados podem trafegar.

O limite máximo da velocidade de transferência de dados imposta pela largura de banda não pode ser ultrapassado. Para modelar este comportamento, impõe-se um limite de performance, que varia de acordo com a utilização das texturas, segundo o qual, se uma aplicação tiver uma performance maior do que este limite, então a performance deve ser diminuída até ficar dentro dos limites impostos pela largura de banda. Este limite é medido para diversos formatos e tamanhos de texturas. Transferências podem ser adicionalmente limitadas por excederem os recursos da hierarquia de memórias cache.

Na seção seguinte será apresentado um simulador como resultado desse modelo de desaceleração proposto.

4.3 Simulador de Performance

Utilizando o modelo de desaceleração proposto anteriormente, um simulador foi implementado (inicialmente em linguagem interna ao programa GnuPlot

(GNUPLOT, 2006), depois em linguagem C) para validação e verificação do modelo. Este simulador foi implementado utilizando-se as premissas e as propriedades do modelo conforme discutido nas seções 4.1 e 4.2.

O simulador agrupa sob um mesmo estágio implementado mais de um estágio lógico descrito para o modelo. Inicialmente, um primeiro estágio calcula a performance básica para uma dada aplicação tomando como entrada parâmetros do hardware e parâmetros iniciais desta aplicação. Os seguintes parâmetros são levados em conta neste primeiro estágio do simulador:

- velocidade do processador de fragmentos
- número de pipelines do processador
- número de ciclos do programa de fragmentos
- número de fragmentos a serem processados

A velocidade e o número de pipelines do processador de fragmentos são dados obtidos da especificação do hardware. O número de ciclos do programa de fragmentos pode ser informado pela aplicação chamada NVShaderPerf (NVIDIA, 2006), desenvolvida pela NVidia ou calculados manualmente quando se conhece as regras de formação de VLIWs. O NVShaderPerf, para um dado programa, produz o número de ciclos que serão necessários para a completa execução do programa de fragmentos pela GPU. O NVShaderPerf informa também a contagem do número de registradores utilizados pelo programa. O NVShaderPerf é bastante efetivo dentro das suas limitações; informando corretamente o número de ciclos para programas de fragmentos que não contém nenhuma espécie de desvio. Para programas que possuem laços ou desvios condicionais, não se deve usar o NVShaderPerf, pois este não implementa corretamente as regras de geração de VLIWs para programas com desvios. Para se obter o número de ciclos para programas com desvios a estratégia é:

- O programa de fragmentos é compilado para obter o equivalente em linguagem de montagem "*assembly*".
- As instruções de desvio e laço são removidas.
- O código sem laços ou desvios é processado pelo NVShaderPerf, que retorna a contagem correta de ciclos sem desvios.
- O número de ciclos é incrementado com a contagem das instruções de laços e desvios. É incluída também a sobrecarga destas instruções de desvio – tabela 2.1.

O primeiro estágio (responsável pela estimativa de performance básica) é implementado como um programa que lê os parâmetros acima mencionados de um arquivo de configuração – figura 4.2. O programa relaciona os parâmetros de acordo com as equações descritas na seção 4.2.1 e gera como saída a primeira aproximação da estimativa de performance.

```
275     // número de ciclos (fora de estruturas if-else-endif) do programa de fragmentos
131072  // número de fragmentos
5.6e9   // número de pipelines de fragmentos * freqüência dos pipelines
6       // custo (em ciclos) da estrutura if-else-endif
0       // número de ciclos dentro do lado if
0       // número de ciclos dentro do lado else
```

Figura4.2: Exemplo do arquivo de configuração do primeiro estágio do simulador.

O segundo estágio do simulador, assim como o primeiro, agrupa mais de um estágio lógico do modelo proposto. Este estágio toma como entrada o valor de performance informado pelo primeiro estágio, o tamanho de triângulos, o tamanho do bloco de coerência e um arquivo com valores de desaceleração previamente calculados. A partir destes dados o segundo estágio utiliza uma interpolação linear para retornar o valor de performance devidamente desacelerado. Os valores previamente calculados de desaceleração são obtidos mediante o ajuste de segmentos de retas sobre as curvas de performance medida versus tamanho de triângulos que é realizado para cada tamanho de bloco de coerência. Este estágio retorna como saída uma segunda aproximação da estimativa de performance, que pode ser utilizada como entrada para o próximo estágio do simulador.

O terceiro estágio do simulador é responsável pela desaceleração devido ao acesso a texturas. Este estágio recebe como entrada a performance estimada pelo estágio anterior, as informações sobre a textura – tamanho e formato. A partir destes parâmetros, o simulador, que tem ajustado um conjunto de retas sobre valores de resposta do hardware para diversos tipos e tamanhos de texturas, verifica se as barreiras do limite de largura de banda não estão sendo ultrapassadas. Calcula-se as penalidades devido a largura de banda, caso necessário, e emite uma nova estimativa de performance para o sistema sendo simulado.

4.4 Características e Limitações

Um modelo de previsão de performance foi proposto escolhendo-se como parâmetros de entrada, propriedades que se julga descrever características importantes do hardware e da carga sendo processada. Para o modelo permanecer simples e expansível, foi formulado em estágios. Os resultados de um estágio deveriam ser independentes dos resultados de outro estágio, para dessa forma permitir que o modelo seja expandido. Mais estágios podem ser adicionados ou os existentes podem ser reformulados e expandidos. A saída de cada estágio já é por si própria uma previsão de performance, de modo que um determinado estágio pode ser completamente desabilitado sem perda de continuidade do processo de previsão de performance.

Um modelo linear permite que funções simples possam ser utilizadas com os valores de entrada. Funções não lineares exigem que mais parâmetros sejam ajustados. Com mais parâmetros necessitando de ajuste, mais imprecisão pode ser adicionada ao funcionamento do modelo. O modelo proposto é estático porque o tempo não é um parâmetro de entrada do modelo.

A idéia de que os estágios do modelo podem ser ortogonais entre si, nos permite a comutação dos estágios. A linearidade do modelo permite que se implemente um simulador utilizando o ajuste apropriado de funções de primeira ordem (retas). O

grau de precisão desejado no modelo varia com o número de retas ajustadas: quanto maior o número delas, tanto maior será a aproximação alcançada pelo modelo.

O modelo depende fortemente da precisão dos parâmetros informados, pois deles advêm os resultados. Se essas informações estão imprecisas, então toda a previsão de performance será imprecisa. Imprecisões podem ser adicionadas em qualquer um dos estágios. Dessa forma, o simulador pode apresentar resultados corretos para um determinado estágio mas não para os demais. Um determinado estágio pode estar mal implementado, não representando corretamente o sistema. Por exemplo, no primeiro estágio não é levada em conta a penalidade devido a utilização de mais do que quatro registradores por fragmento em cada programa de fragmentos.

5 Validação e Verificação do Modelo

O capítulo anterior apresentou um modelo de performance de hardware gráfico que foi implementado na forma de um simulador. Da mesma forma que o modelo, o simulador é composto de estágios que são agrupados para obtenção de estimativas de performance de uma GPU. Este capítulo analisa a validade do modelo em relação a experimentos sintéticos e verifica as estimativas frente a uma aplicação complexa. Validação significa a elaboração de experimentos para os quais se conhece os resultados (ou pode-se medí-los com relativa facilidade) e a analise de quão próxima fica a resposta do simulador nestes experimentos. Por exemplo, dados programas de fragmentos para os quais se conhece o número de ciclos para execução de cada fragmento, analisa-se a resposta do simulador para cada um dos casos. O modelo/simulador é válido se os resultados estiverem dentro da margem de erro determinada como aceitável. Para verificação, os resultados do simulador são comparados com medidas de performance de uma aplicação complexa para a qual a performance não é facilmente determinável. Foi determinado como margem de erro aceitável 10% em cada estágio. Cada estágio do simulador contribuindo com cerca de 10%, o erro acumulado ao final de três estágios em série é esperado ser da ordem de 33%.

Para validar e verificar o modelo e a implementação do simulador desenvolvido neste trabalho foram escolhidos dois hardwares como referência: a placa gráfica Geforce 7800 GTX com GPU G70 da Nvidia (NVIDIA, 2005) e a placa gráfica Radeon X1900 XTX com GPU R580 da ATI (ATI, 2006). Estes modelos foram escolhidos por terem a capacidade de executar programas de fragmentos que contêm instruções de desvio de fluxo de execução.

Ao longo deste capítulo medidas de performance em ambas as placas serão comparadas com os resultados obtidos do simulador. Nas seções 5.1.1 à 5.1.5 as medidas serão realizadas utilizando situações sintéticas. Na seção 5.2 os resultados do simulador serão comparados com os dados de uma aplicação real.

O objetivo é comparar os resultados de uma configuração do simulador para estimativa de performance com o desempenho de um hardware gráfico real submetido a diversas cargas, sintéticas e reais. Os ambientes de medida são apresentados, assim como as medidas que foram realizadas e os resultados que foram obtidos.

5.1 Validação do Simulador com Experimentos Sintéticos

O simulador foi validado comparando valores calculados pelo simulador e valores medidos para experimentos sintéticos simples. Foram utilizadas situações

sintéticas, para manter a variação dos parâmetros a serem avaliados sob controle. Utilizou-se pequenos núcleos de programas de fragmento com comportamento e performance conhecidos e variou-se os demais parâmetros de entrada. Os parâmetros analisados foram:

- número de ciclos para execução de cada fragmento [ciclos/fragmento]
- número de fragmentos por quadro [fragmentos/quadro]
- tamanho dos triângulos por quadro [fragmentos/triângulo]
- tamanho do bloco de coerência [fragmentos/bloco coerente]
- tamanho das texturas acessadas [texels por textura]
- formato das texturas [bytes por texel]

Os resultados obtidos da estimativa de performance do simulador são então comparados com medidas realizadas no hardware escolhido como referência. Esses resultados devem ser expressos na mesma unidade de medida para fins de comparação. A unidade utilizada é FPC, Frames Per Clock, o número de quadros ("*frames*") desenhados a cada ciclo do relógio da GPU ("*clock*"), que é o número de FPS (Quadros Por Segundo) – conforme descrito no capítulo 3 – dividido pela freqüência do processador da GPU.

A técnica utilizada para a coleta destes tempos é a mesma descrita na seção 3.2, com código de CPU instrumentado. A aplicação usa a API Direct3D para ativar um programa de fragmentos na GPU. Esta aplicação é responsável por enviar instruções para o hardware gráfico e medir o tempo gasto por essas instruções. A aplicação também é responsável por configurar corretamente o estado do hardware gráfico. Por exemplo, carregar os programas de fragmentos a serem utilizados e também informar a geometria a ser desenhada na tela, indicando as posições dos vértices utilizados. É a aplicação que carrega as texturas que podem ser utilizadas pelos programas de fragmentos. O programa de fragmentos pode ou não utilizar estas texturas.

A placa de video da NVidia GeForce 7800 GTX possui as seguintes características:

- Freqüência do processador: 486 MHz
- Número de pipelines de fragmentos: 24

A placa de video da ATI X1900 XTX possui as seguintes características:

- Freqüência do processador: 500 MHz
- Número de pipelines de fragmentos: 48

Estas características básicas do hardware servem como guia genérico sobre a capacidade máxima de processamento que a GPU pode oferecer. A capacidade máxima de processamento de fragmentos da GPU indica quantos fragmentos o processador está apto a processar por segundo. Esta capacidade máxima é dada pela freqüência de operação do processador de fragmentos e pelo número de pipelines:

$$\text{Perf\_Max}(\mathit{freq}, \mathit{pipes}) = \mathit{freq} * \mathit{pipelines} = \frac{\mathit{fragmentos}}{s} \quad (5.1)$$

onde *freq* é a freqüência de operação do processador e *pipelines* é o número de pipelines presentes no processador que indica quantos fragmentos podem ser operados simultaneamente pela GPU – ou seja, esse valor tem unidade de fragmentos. Substituindo estes valores pelos valores da 7800 GTX, temos

$$\text{Perf\_Max}(\mathit{freq}, \mathit{pipes}) = 486.000.000 * 24 = 11{,}664 \cdot 10^9 \frac{\mathit{fragmentos}}{s} \quad (5.2)$$

indicando que o hardware de referência da NVidia pode processar 11,6 bilhões de fragmentos a cada segundo. Na equação (5.3) é assumido que cada fragmento necessita de 1 ciclo para ser processado.

Para a X1900 XTX temos:

$$\text{Perf\_Max}(\mathit{freq}, \mathit{pipes}) = 500.000.000 * 48 = 24 \cdot 10^9 \frac{\mathit{fragmentos}}{s} \quad (5.3)$$

ou seja, o equipamento da ATI pode processar aproximadamente o dobro de fragmentos por unidade de tempo do que o modelo da NVidia. No entanto, vamos notar nos resultados (gráficos) que nem sempre é possível atingir o dobro da performance da 7800 com a X1900

Os demais parâmetros a serem analisados são responsáveis por diminuir a performance máxima. O impacto desses parâmetros na performance são aproximados pelos estágios posteriores do simulador.

Uma série de testes de validação foi realizada para avaliar o simulador de previsão de performance. Estes testes avaliam individualmente cada um dos estágios do simulador discutidos na seção 4.2. Um ou mais testes foram realizados para cada estágio do simulador. O primeiro estágio a ser avaliado foi o do simulador de performance básico. Para este estágio foram realizados dois testes variando a carga sobre o processador de fragmentos: um variando a quantidade de fragmentos processados e outro variando a quantidade de ciclos necessários para processar cada fragmento.

O segundo estágio, responsável pela redução de performance relativa ao tamanho de triângulos utilizados e a freqüência com que desvios no código são tomados, foi avaliado utilizando três testes: O primeiro teste compara a performance quando o tamanho de triângulos é variado. O segundo teste mostra a variação da performance quando a freqüência de tomada de desvios muda.

O estágio responsável pela redução de performance devido a acessos a texturas é avaliado por fim. Para avaliação desse estágio, utilizou-se de programas de acesso a texturas com acessos independentes. Foi realizada a leitura do valor de textura variando o tamanho da textura, em texels, e o formato da textura, em bytes por texel.

Para cada estágio procurou-se a meta de manter o erro relativo abaixo de 10%. O erro relativo dos estágios compõe-se ao longo da simulação. De forma que se cada estágio contribui com cerca de 10% de erro, é esperado que ao final da simulação se tenha um erro composto de no máximo 33%, para estes estágios em série com erro menor ou igual a 10% por estágio.

5.1.1 Simulador Básico

Conforme visto na seção 4.2.1, o primeiro estágio do simulador é responsável por uma estimativa da performance básica entregue pela placa gráfica, conhecidas algumas características do hardware (velocidade de operação e número de pipelines) e a carga à qual o hardware estará submetido:

$$\text{Perf} = \frac{\text{Perf\_Max}}{\text{Carga}} \tag{5.4}$$

onde Perf_Max foi definido na equação (5.2) e carga é dada pelo número de fragmentos multiplicado pelo número de cíclos por fragmento. Mais formalmente, carga é definida por:

$$\text{Carga} = \text{Numero de Fragmentos} * \text{Ciclos por Fragmento} \tag{5.5}$$

A primeira parte dos testes de validação realizados compara os resultados previstos pelo simulador básico com dados medidos variando parâmetros de entrada do hardware de referência e do programa executado no processador de fragmentos – figura 5.1 – que são:

Parâmetros de hardware:

- Freqüência do processador de fragmentos
- Número de pipelines de fragmentos

Parâmetros de software:

- Número de ciclos que compõem o programa de fragmentos
- Número de fragmentos processados

Dois experimentos foram realizados para a validação do primeiro estágio do simulador: um experimento variando o número de fragmentos a serem processados e o outro variando o número de ciclos que compõem um programa de fragmentos. A freqüência do processador e o número de pipes não foram variados por serem parâmetros intrínsecos do hardware – em geral, não variam durante a execução de uma aplicação.

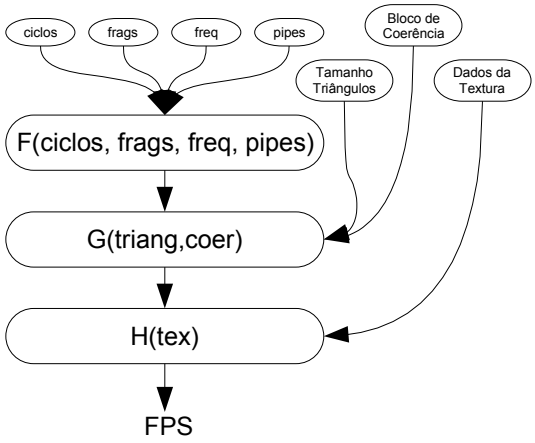

Figura 1: Estágios implementados pelo simulador

No primeiro experimento foi variado o número de fragmentos que compõem cada quadro. Para cada quadro, um retângulo é desenhado na tela utilizando dois triângulos – cada triângulo com metade dos fragmentos a serem processados. Para variar o número de fragmentos, a resolução da tela (no modo tela-cheia) foi variada desde a resolução 800x600 até 1600x1200, o máximo permitido pela configuração do sistema (combinação da placa aceleradora gráfica com monitor). A API limita as possíveis resoluções de tela quando se está trabalhando no modo tela-cheia. A aplicação ficou limitada às resoluções oferecidas pelo sistema. Para esta medição foi utilizado sempre o mesmo programa de fragmentos, o qual não possui desvios de execução e foi composto de 10 ciclos sem acesso a texturas – figura 5.2.

A construção do programa de fragmentos deve evitar possíveis otimizações por parte do compilador/otimizador. Do contrário, o número de ciclos por fragmento poderá não ser o esperado. Para evitar este problema, utiliza-se o programa NVShaderPerf da Nvidia (NVIDIA, 2006), para determinar o número exato de ciclos por fragmento.

É importante notar que se o programa de fragmentos tem poucas instruções, podem ocorrer flutuações de performance nos dados medidos. Em alguns experimentos foram observadas variações de mais de 100% entre execuções de programas com poucas instruções. As flutuações de performance são causadas quando a carga a ser medida é menor ou da mesma ordem da carga do sistema (CPU, sistema operacional, outras aplicações, etc.).

```
ps_3_0

  dcl_color v0
  dcl_texcoord0 v1

  def c0, 0.50, 0.50, 0.00, 0.00
  def c1, 0.00, 0.50, 0.00, 0.00

  add r0, c1, c1
  add r1, c0, c0
  mul r0, r0, r1
  add r0, r0, v0
  sincos r1.x, r0.y
  add r0, r1, v0
  sincos r1.x, r0.y
  add r0, r1, v0
  ....                // Repetir n vezes as duas instruções anteriores
mov oC0, r0
```

Figura 2: Exemplo de código de processador de fragmentos

Para o experimento de análise de performance em função do número de fragmentos – figura 5.3, foi observado erro relativo menor do que 2% para 7800 com número de fragmentos acima de 900.000. Flutuações no sistema se tornam perceptíveis devido à incapacidade da carga de trabalho passada à GPU amortizar as demais variações de carga que ocorrem. Conforme a carga de trabalho sobre a GPU vai aumentando, esta entra em um regime de trabalho mais estável e consegue amortizar

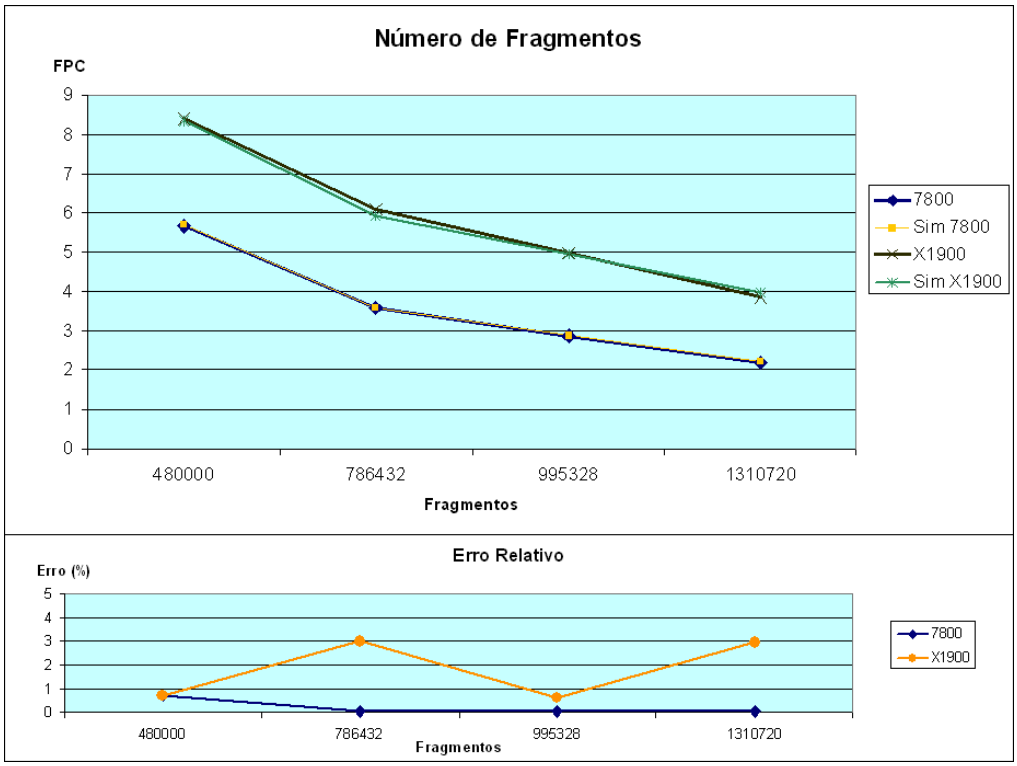

Figura 3: Variação da Performance como função do número de fragmentos processados

melhor as flutuações de performance do resto do sistema (CPU, por exemplo).

O segundo experimento para validar o simulador básico, variou o número de ciclos por fragmento (isto é, o programa foi alterado), mas manteve-se fixo o número de fragmentos (1024x768). Os demais parâmetros também permaneceram fixos. Para variar o número de instruções em cada programa foram utilizados diferentes núcleos de

programas de fragmento, cada um com um número diferente de instruções – figura 5.4. Cada programa consiste em uma repetição de instruções que o otimizador não consegue agrupar. Para o hardware da NVidia foi utilizado o NVShaderPerf (NVIDIA, 2006) para indicar o número correto de ciclos em cada programa. Para o hardware da ATI, foi utilizado um método analítico de contagem do número de instruções MAD que compõem o programa, já que se sabe que o processador de fragmentos da GPU da ATI pode realizar apenas uma instrução MAD por ciclo – conforme visto no capítulo 2.

```
ps_3_0

  dcl_color v0
  dcl_texcoord0 v1

  def c0, 0.01, 0.01, 0.01, 0.01
  def c1, 1.00, 1.00, 1.00, 1.00

  mov r1, v0
  mov r2, v1
  mad r0, c1, r1, r2
  mad r0, r0, r2, r2
  mad r2, r2, v0, r0
  mad r0, r0, r2, r2
  mad r2, r2, v0, r0

mov oC0, r2
```

Figura 4: Exemplo de programa de fragmentos com 6 ciclos no hardware da ATI

A figura 5.5 mostra o resultado do simulador com relação à performance medida nas placas de vídeo para os experimentos variando o número de ciclos por fragmento e mantendo todos os outros parâmetros constantes. Para programas com poucos ciclos – menos de 5 – o erro relativo aumenta. Quando a carga sobre a GPU aumenta, aumentando o número de instruções, o hardware entra em um regime de trabalho mais estável, onde as flutuações devido a outras cargas do sistema são amortizadas pelo tempo necessário para execução do programa de fragmentos.

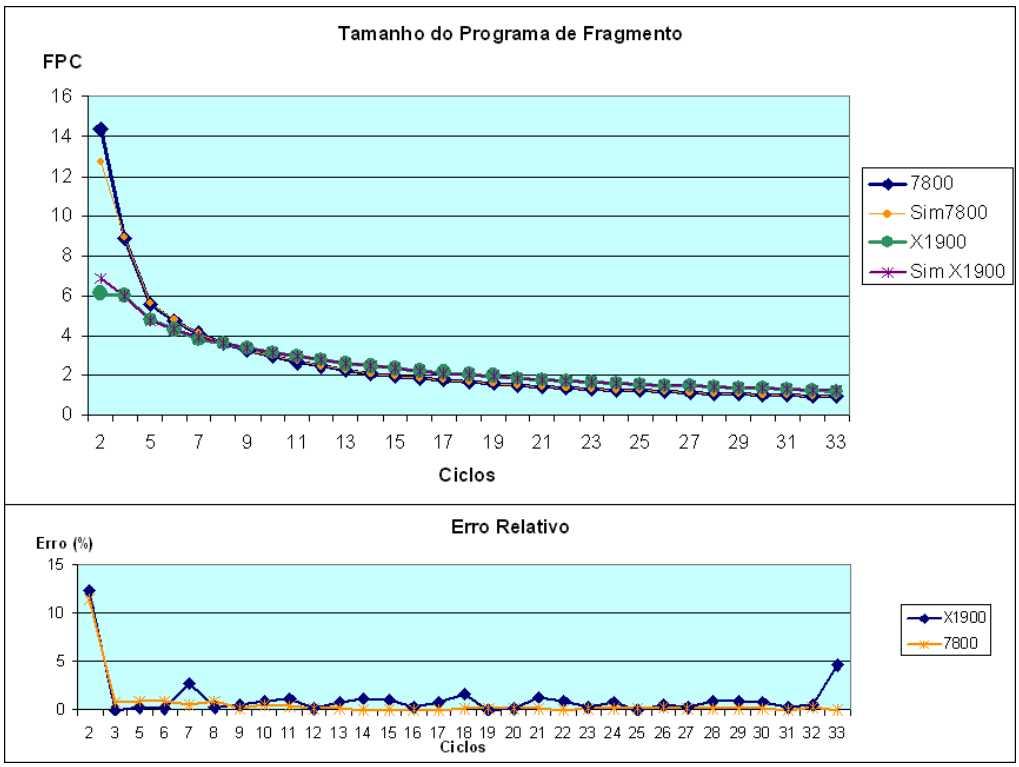

Figura 5: Variação da performance com o número de ciclos

5.1.2 Tamanho de Triângulos

O segundo estágio do simulador é responsável por considerar a redução de performance devido ao tamanho de triângulos. Para validar este módulo foi construído um experimento variando o tamanho dos triângulos desenhados. O número total de fragmentos processados e o número de ciclos nos programas de fragmentos foram mantidos constantes.

Para o experimento de medição de performance com relação ao tamanho dos triângulos, a imagem desenhada na tela, em modo tela-cheia na resolução 1600x1200, é um retângulo cobrindo toda tela – ou seja, número constante de fragmentos por quadro. Inicialmente, esse retângulo é composto de 2 triângulos – figura 5.6 lado esquerdo. A seguir, esse retângulo passa a ser composto por oito triângulos – figura 5.6 lado direito. Isto é, o retângulo passa a ter dois triângulos de lado, em seguida quatro triângulos de lado e assim por diante. Como o número de fragmentos compondo a imagem final é constante, o número de fragmentos por triângulo diminui conforme aumenta o número de triângulos compondo o retângulo. A relação do número de fragmentos por triângulos é dada por:

$$\text{frags\_por\_tri} = \frac{\textit{número de fragmentos por quadro}}{\textit{número de triângulos por quadro}} \qquad (5.6)$$

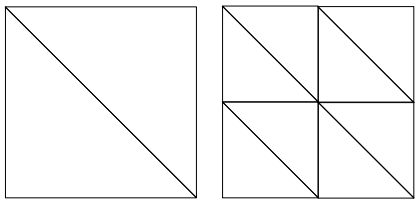

Figura 6: Variação do tamanho de triângulos para preencher uma tela inteira.

O tamanho do triângulo foi variado desde 240.000 fragmentos até 5 fragmentos por triângulo. Nota-se uma redução de performance acentuada quando o triângulo tem tamanho menor que 58 fragmentos. Isto sugere que o hardware é otimizado para triângulos de tamanho igual ou maior que 58 fragmentos. Note que um triângulo de 58 fragmentos equivale aproximadamente a metade da área de um quadrado de 10x10 pixels – ou seja, 0,007% da área de um modo de baixa resolução com 1024x768. A figura 5.7 apresenta os valores de performance medidos juntamente com os resultados do simulador.

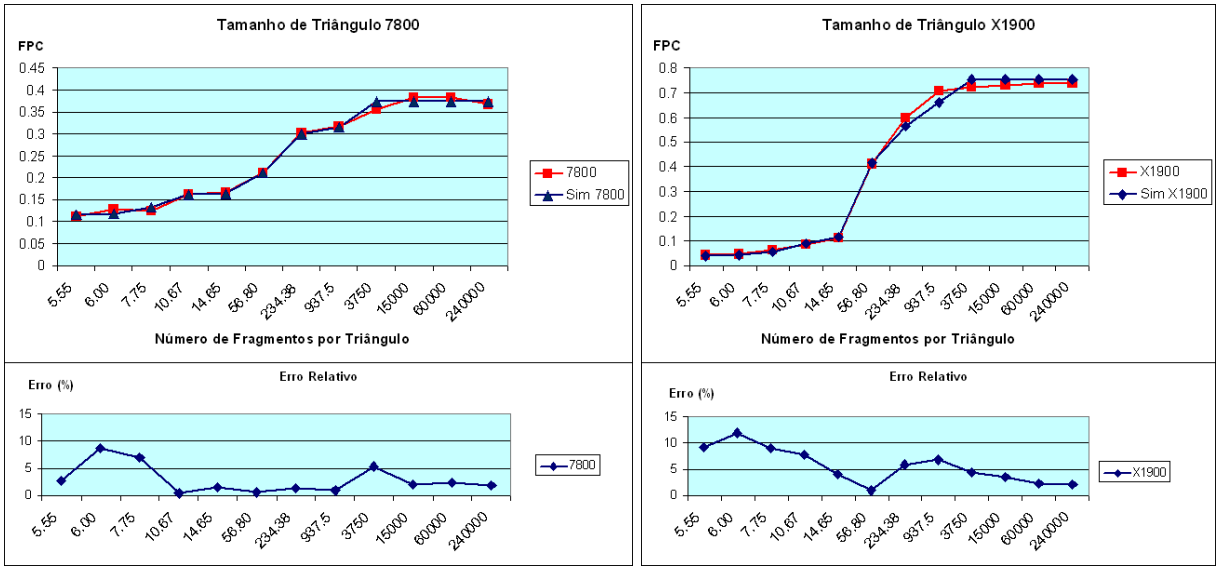

Figura 7: Variação de performance com o tamanho do triângulo. Valores medidos, valores simulados e erro relativo.

5.1.3 Blocos de Coerência

O segundo estágio do simulador também é responsável por considerar a redução de performance ocasionada pelo tamanho do bloco de coerência (seção 4.2.4) nas tomadas de desvio de código. O experimento consiste em avaliar a performance de execução de um programa com uma estrutura if-then-else. O número de fragmentos (1600x1200) e triângulos (2) são mantidos constantes. A variável do experimento é a freqüência com que o if e o else são tomados para os fragmentos ao longo da imagem.

A figura 5.8 ilustra o tipo de programa de fragmentos utilizado. O programa de fragmentos varia a freqüência com que os desvios de execução são tomados dividindo a tela em colunas verticais. A condição para decisão de tomada do desvio é baseada na coordenada X de textura dos fragmentos. Todos os fragmentos pertencentes a uma mesma coluna executam um lado do desvio condicional. Colunas de largura maior levam a blocos de coerência maiores – figura 5.9. Colunas de largura menor levam a blocos de coerência menores (o agrupamento de fragmentos que decidem ir por um lado ou por outro do desvio se torna menor). Fragmentos pertencentes a uma determinada coluna tomam um lado do desvio. Fragmentos pertencentes às colunas complementares executam o outro lado do desvio. As instruções a serem executadas em um ou outro lado do desvio são as mesmas. Para obter números diferentes de colunas, apenas uma constante no programa precisa ser alterada. Ou seja, o número de ciclos (ou VLIWs) para execução do programa é constante. Note que todos os outros parâmetros do simulador são mantidos constantes.

Sem a presença de desvios de execução, note que mediríamos performance constante para este experimento. No entanto, observamos queda de performance quando o bloco de coerência é menor do que um determinado valor (dependente da GPU).

Para o hardware da NVidia, o número de colunas variou desde 2 até 40 colunas com um passo de 2 colunas, para uma tela de resolução de 1600x1200 pixels. Para o hardware da ATI foi necessário utilizar um número maior de colunas para se observar a saturação no ganho de performance devido a desvios dinâmicos. Para a placa da ATI o número de colunas variou de 2 até 300.

É possível verificar nos experimentos que a performance aumenta devido à utilização de desvios dinâmicos acima de uma determinada largura da coluna de fragmentos – lado esquerdo nos gráficos da figura 5.10. No hardware da NVidia esse valor onde os efeitos da utilização de desvios começa a se notar é atingido quando 26 ou menos colunas são utilizadas para cobrir a tela. No hardware da ATI é necessário haver 212 colunas. Como a largura de uma coluna é dada por:

$$L = \frac{largura\ da\ tela}{número\ de\ colunas} = \frac{pixels}{colunas} \quad (5.7)$$

```
ps_3_0
  dcl_color v0
  dcl_texcoord v1

  def c0, 0, 0.5, 1, 0
  def c1, 1, 0, 0, 0

  mov r0, v0

  // Armazena em r2 a parte fracionaria da multiplicação
  mul r1.x, v1.x, c1.x
  frc r2.x, r1.x
  ...
  if_ge r2.x, c0.y    // Tomada de decisão
    ...               // Um lado do desvio
  else
    ...               // Outro lado do desvio
  endif

mov oC0, r0
```

Figura 8: Exemplo de código com desvio condicional que divide os fragmentos em 2 grupos (coordenada x de textura maior ou igual que 0,5 e coordenada de textura menor que 0,5).

Para a placa da NVidia (7800) o ponto a partir do qual se tira proveito dos desvios de código dinâmicos é quando a largura das colunas é igual ou maior do que 60 pixels por coluna. Para a placa da ATI (X1900), o ponto a partir do qual se tira vantagem dos desvios de código dinâmico é quando a largura das colunas é maior ou igual a 8 pixels por coluna. Esta característica pode ser explicada da seguinte maneira (acompanhe na figura 5.10).

Quando a largura da coluna é a maior possível (512 pixels) a coerência para a decisão da tomada de desvio é máxima. Nesta situação, grandes blocos de fragmentos decidem ir pelo mesmo lado do desvio. Quando todos os fragmentos que estão num pipeline decidem executar o mesmo lado de um desvio do código, o pipeline descarta a execução do outro lado do desvio. Quando um trecho de código pode ser descartado, a performance de execução do código aumenta. Quando todos os fragmentos da tela executam apenas um lado do desvio (isto é, apenas o if ou apenas o else), obtém-se performance máxima.

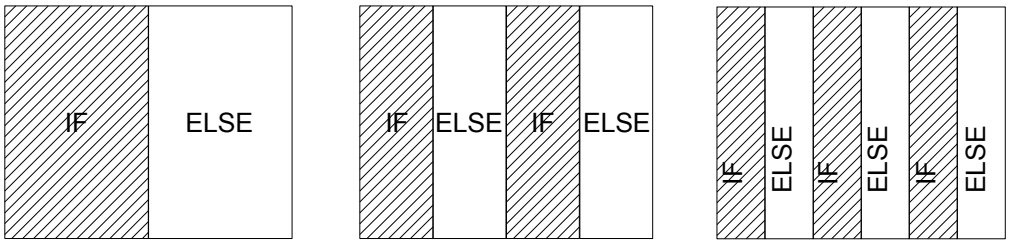

Figura 9: Variação da largura da coluna, que determina o tamanho do bloco de coerência na execução de um programa com estrutura if-then-else.

Conforme a largura da coluna vai diminuindo, pipelines começam a ter uma mistura de fragmentos. Alguns desses fragmentos decidem por um lado do desvio, outros decidem por outro lado do desvio. Neste caso, cada pipeline SIMD não pode descartar a execução de um dos lados do desvio. Para resolver a situação, o pipeline executa um lado do desvio e os fragmentos que decidiram por este lado atualizam seus registradores de saída, os demais fragmentos descartam o processamento e não

atualizam seus registradores de saída. Em seguida, o pipeline executa o outro lado do desvio para todos os fragmentos e permite que os fragmentos que não atualizaram seus registradores de saída na execução anterior o possam fazer nesta execução: os fragmentos que atualizaram seus registradores de saída na execução anterior não têm os registradores de saída atualizados nesta execução. Ou seja, o custo de execução de cada fragmento é a soma dos custos do if e do else (incluindo os custos adicionais que as instruções de controle de fluxo acarretam). Dessa forma o pipeline terá o trabalho de executar os dois lados de um desvio. No entanto, nem todos os pipelines precisam estar executando os dois lados do desvio. Pipelines que têm fragmentos de apenas um dos lados dos desvios continuam executando apenas um lado do desvio. Por isso, a performance ainda não é a menor possível, pois alguns pipelines ainda descartam código, havendo uma mistura dos que descartam código e dos que não descartam.

Quando a largura da coluna diminui abaixo de um valor limite, todos os pipelines estão executando os dois lados do desvio, de forma que todos os pipelines tem a performance reduzida. Esta é a situação em que a performance é a menor possível, quando todos os pipelines tem que executar todo o código do programa, porque contêm pelo menos um fragmento de cada lado do desvio. A partir desse ponto, por menor que seja a largura da coluna, a performance não vai se degradar mais, pois não importa o número de fragmentos que necessitam ir para um lado ou para o outro do desvio, todos os pipelines vão executar os dois lados do desvio.

Nos gráficos da figura 5.10 podemos ver a performance máxima e mínima para o mesmo programa com desvios. Quando a performance é máxima (ideal) apenas o bloco do if ou apenas o bloco do else estão sendo executados. Com a performance mínima, o programa sempre executa tanto o if quanto o else. Esta á a performance para GPUs sem desvio dinâmicos (por exemplo, NV3X ou R4x0). Isto é, para APIs anteriores à DX9c que introduziu o PS3.0 (Pixel Shader 3.0). GPUs sem desvios dinâmicos sempre caem no pior caso de performance, pois sempre precisam executar os dois lados de um desvio.

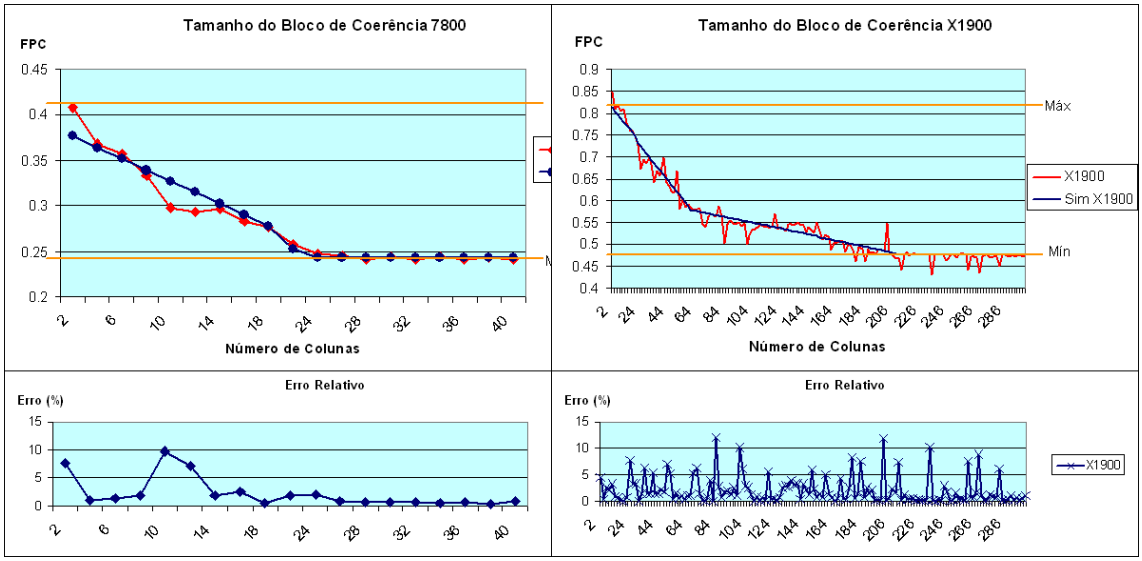

Figura 10: Variação da performance com o tamanho da coerência da tomada de desvios. A performance diminui (o tempo aumenta) conforme o tamanho do bloco de coerência diminui. Abaixo de 60 fragmentos por coluna a performance é constante.

Para a placa Geforce 7800 (G70) o ganho pela utilização de desvios começa a ocorrer quando a largura da coluna é maior ou igual a 60 fragmentos (24 colunas na figura 5.10). Para a placa de vídeo da ATI (R580) este ponto é alcançado quando a largura da coluna é maior ou igual a 8 fragmentos (212 colunas na figura 5.10). Isto sugere que diferenças de implementação dos pipelines de fragmentos, discutidas no capítulo 2, permitem ao hardware da ATI explorar de forma mais eficiente a utilização de desvios dinâmicos de fluxo no programa de fragmentos.

5.1.4 Acesso a Texturas Independentes

O terceiro estágio do simulador utilizou um módulo para acessos a texturas (ou memória). Os efeitos observados aqui são governados pela habilidade da GPU de mascarar o custo de acessos a memória. A GPU tem a habilidade de mascarar os acessos a memória utilizando níveis de memória cache. A GPU despacha uma série de comandos de acesso a memória assíncronos enquanto realiza outras operações não dependentes para manter o pipeline operando durante o tempo que a operação de acesso a memória leva para retornar – capítulo 2. Muitos acessos a texturas diminuem as chances do processador mascarar a latência da memória. A ferramenta GPUBench (BUCK, 2005) reporta que o custo de acesso a texturas dependentes é governado inteiramente pela coerência dos acessos a memória (FATAHALIAN, 2004). O perigo, portanto, com leitura de texturas dependentes é a sua habilidade de criar padrões de acesso incoerentes para a cache (HALL, 2003). Quando uma cache não pode suprir as necessidades de leitura de memória, uma bolha é introduzida no pipeline e a performance é reduzida.

Para acelerar o processamento de instruções que fazem acesso a texturas, cada pipeline é equipado com uma memória cache, que pode armazenar alguns dados de memória acessados mais freqüentemente. Cada pipeline de fragmentos possui uma cache de primeiro nível (L1) e o sistema como um todo tem uma cache de segundo nível (L2). Todas as memórias cache L1 se reportam à única L2, quando os dados necessários não estão presentes na L1. A distância temporal da L2 à GPU é superior à da L1. A L2 se reporta ao terceiro nível da hierarquia de memória que é a memória de toda placa de vídeo (o framebuffer). O framebuffer é o maior e mais distante temporalmente de todos os níveis de memória.

O experimento de acesso a textura utiliza um programa de fragmentos que possui uma única instrução que realiza um acesso a textura. A cena consiste de dois triângulos formando um quadrilátero que cobre a tela inteira (de resolução 1600x1200). Utilizou-se um número reduzido de triângulos, pois como já foi visto, se triângulos muito pequenos são utilizados, a performance pode ser comprometida. O tamanho da textura foi variando desde 2x2 texels até o limite permitido por ambas placas gráficas de 4096x4096 texels em potências de 2. Apesar do hardware da NVidia permitir a utilização de texturas com dimensões que não são potência de 2, optou-se pela utilização das mesmas dimensões em ambos equipamentos.

Além da variação do tamanho em texels das texturas, variou-se o formato de armazenamento, o que altera a quantidade de dados que precisam ser transferidos por acesso e armazenados também. Usou-se dois formatos de inteiros de 8 bits: o formato DXT1, que comprime a textura e o formato sem compressão ARGB8. Ambos formatos guardam 8 bits por canal de cor em cada texel mas o formato DXT1 comprime os dados para minimizar o uso de largura de banda na transferência. Além destes dois formatos

de 8 bits por canal, foram utilizados dois formatos de ponto flutuante: ponto flutuante de 16 bits por canal, ARGB16 e ponto flutuante de 32 bits por canal, ARGB32. Estes formatos guardam, respectivamente, 16 e 32 bits por canal de cor, sem compressão. A figura 5.11 apresenta os resultados para a variação das texturas em formatos DXT1, ARGB8, ARGB16 e ARGB32.

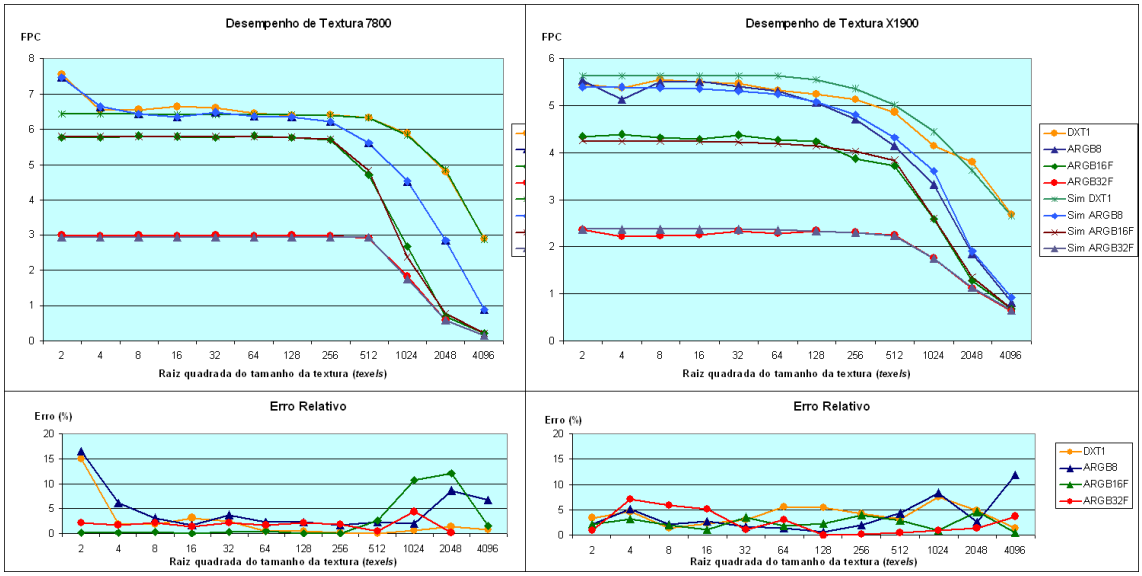

Figura 11: Performance como função do tamanho da textura

Note nos gráficos da figura 5.11 que, para um determinado formato de textura, há sempre um patamar de performance máxima (à esquerda dos gráficos). Este patamar indica a performance máxima que se pode obter devido à transferência de dados (saturação da largura de banda) – isto é, depende da largura do barramento de memória, da freqüência do relógio de memória e da quantidade de informação que precisa ser transferida por acesso (bytes por texel). À direita do patamar há sempre perda de performance, devido às caches não conseguirem suprir os dados à taxa máxima. O simulador aplica a limitação de performance, para os casos que se encontram no patamar correspondente ao formato de textura em uso, e realiza a redução de performance para os casos em que o tamanho da textura e formato estão fora do patamar de performance máxima.

5.1.5 Acesso a Texturas Dependentes

O desenvolvimento do simulador foi iniciado numa placa gráfica GeForce 6800 GT Ultra da NVidia, equipada com uma GPU NV40. Um experimento que envolveu bastante tempo e análise foi o de acessos a texturas dependentes. O experimento e os dados apresentados nesta seção (e apenas nela) foram obtidos na GeForce 6800. É importante notar que esta é outra implementação possível para o terceiro estágio do simulador.

O experimento de acesso a textura dependente utiliza um programa que realiza sucessivos acessos a texturas com uma cena que consiste de dois triângulos formando um quadrilátero que cobre a tela inteira (de resolução 1024x768). O programa de fragmentos utiliza acessos dependentes a texturas onde os endereços (texels) acessados em uma segunda textura são indicados por valores lidos de uma outra textura. A

primeira textura acessada (que rege o acesso aos endereços de memória) é chamada de "textura independente" – figura 5.12 – e a segunda textura acessada é de "textura dependente". Os endereços de acesso para a segunda textura são dados pelos valores lidos da textura independente, isto é, resultado = textura2 (textura1()). Os valores lidos da textura dependente não influenciam a performance da GPU – apenas o tamanho e o padrão de acesso da textura dependente determinam a performance do experimento.

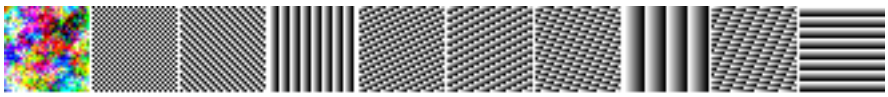

Figura 12: Padrões de acesso utilizados (da esquerda para direita): aleatório, xadrez, step2, step3, step4, step5, step6, step7, step8, step3_rot

O experimento consiste em analisar a performance de execução do programa com acessos a texturas em função do tamanho das texturas. O tamanho da textura independente (a primeira a ser lida pelo programa) é mantido constante e muito pequeno (32x32 texels) para estar sempre garantidamente em cache. O tamanho das texturas dependentes variou de 2x2 a 2048x2048 texels, de modo que se pudesse detectar as quebras de cache. Ambas texturas utilizaram formato RGBX de 8 bits por canal. O tamanho das texturas dependentes foi variado uniformemente de 2 em 2 texels em cada dimensão entre os dois extremos. Como nos demais testes, o número de fragmentos (1024x768) e triângulos (2) foi mantido constante e 20 medidas foram realizadas com uma amostra a cada segundo. Esse processo foi repetido para cada uma das texturas dependentes, mantendo-se fixa a textura independente.

Os dados estão representados em um gráfico "tempo de desenho" versus "tamanho da textura dependente" – figura 5.13. Através desse gráfico de performance pode-se observar o impacto do tamanho da textura no desempenho do processador de fragmentos. A partir da forma do gráfico é possível estimar quando a L1 não conseguirá suprir as necessidades do pipeline para acessos às texturas e quando nem mesmo a L2 não conseguirá atender a estas necessidades.

O programa que executa no processador gráfico instrui o processador de fragmentos a ler o valor de cor da textura independente nas coordenadas indicadas pelo

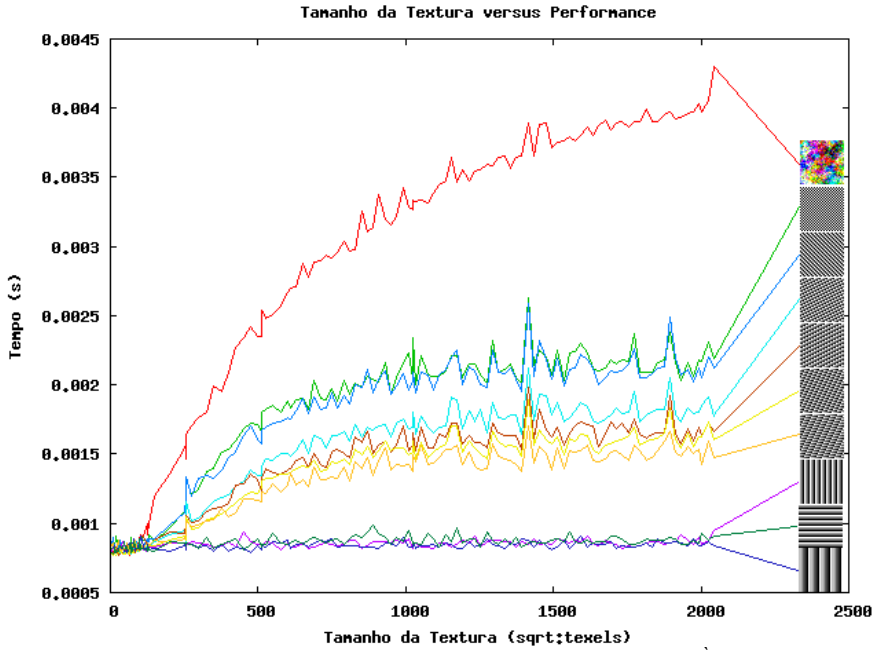

Figura 13: Performance versus tamanho da textura dependente. À direita estão as texturas independentes, que regem o padrão de leitura da memória.

processador de vértices e armazenar esse resultado num registrador temporário r0. As coordenadas de textura são normalizadas em 1, isto é, seus valores vão de (0,0) até (1,1). Como as cores armazenadas nas texturas também são normalizadas em 1, pode-se utilizar o valor de uma cor para endereçar uma posição em uma textura. A textura independente, por ser de reduzidas dimensões deve ficar inteiramente alocada na cache L1 e portanto o acesso a ela não deve piorar a performance. Em seguida o valor armazenado em r0 é utilizado para a carga da textura dependente para o registrador de cor do fragmento. A textura dependente tem o seu tamanho variado a cada execução do aplicativo. Espera-se que a partir de um determinado tamanho o acesso a ela comece a diminuir a performance, pois ela não estará mais inteiramente contida na L1 e mais adiante nem mesmo na L2.

O padrão de acesso à memória é regido pela textura independente, que informa quais endereços de textura devem ser lidos da textura dependente. Um padrão de acessos mais (ou menos) coerente, leva a desempenho melhor (ou pior). Dentro do intervalo de variação do tamanho de texturas dependentes existem duas regiões distintas, ao longo da coordenada de tamanho, para o comportamento do hardware – figura 5.14:

Primeira Região (à esquerda no gráfico da figura 5.14) - Compreende texturas de tamanho 2x2 texels até cerca de 100x100 texels; Nesta região, tanto a L1 quanto a L2 conseguem suprir as necessidades do *pipeline*, não existindo portanto perda na performance. Esta região (mais a esquerda no gráfico da figura 5.14) é representada no módulo do simulador por uma reta quase horizontal, pois a performance supõe-se estar no máximo possível para o programa de fragmentos sendo utilizado. Esta reta se prolonga até à segunda região, quando a curva de performance se altera.

Segunda Região (à direita no gráfico da figura 5.14) - Compreende texturas de tamanho maior que 100x100 texels; Acredita-se que nessa região a L1 não consegue mais suprir as necessidades do *pipeline*, existindo portanto uma redução de performance tanto maior quanto maior for o tamanho da textura dependente – ou seja, o aumento do tamanho da textura dependente causa mais substituições na cache. No modelo isso é representado por uma reta de inclinação positiva. Quanto maior a inclinação desta reta, mais rapidamente a L1 está sendo exaurida. Essa inclinação depende do padrão de acesso à textura, isto é, depende do tipo da textura dependente.

Para texturas de tamanho maior do que 600x600 texels acredita-se que tanto L1 quanto L2 podem não estar ajudando a performance do *pipeline*. Esta região é representada no modelo por uma terceira reta, que se inicia onde a segunda termina. Nesta região está ocorrendo uma combinação de redução de performance devido a L1 e a L2. A saturação que se vê (pelo menos para as curvas verticalmente do meio do gráfico da figura 5.14) parece indicar que o sistema (GPU) atingiu um "equilíbrio" e

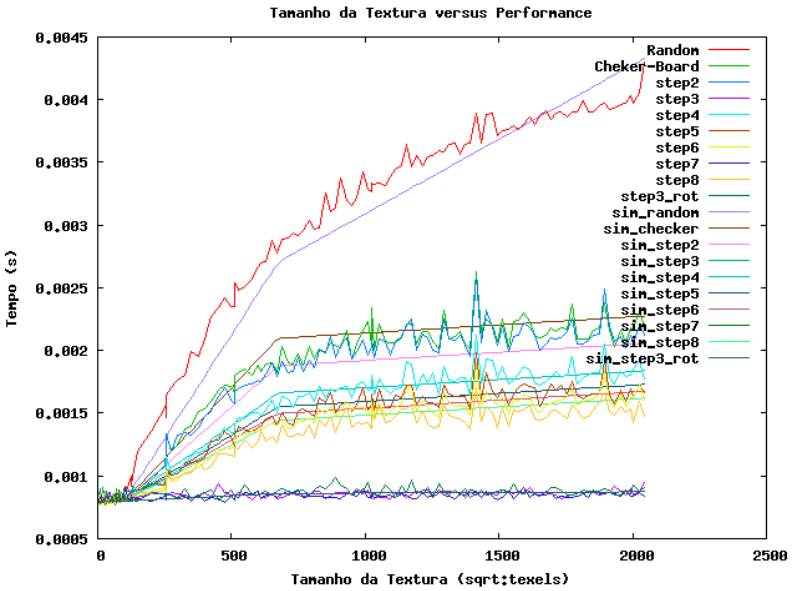

Figura 14: Performance Medida e Performance Estimada pelo Simulador

está conseguindo manter a mesma declividade da reta. Quanto mais altas as freqüências na textura independente, mais freqüentemente se quebra as caches e mais difícil se torna o retorno à situação de "equilíbrio".

Os valores armazenados nas texturas independentes são de especial interesse, pois controlam o padrão de acesso às texturas dependentes. As texturas independentes são todas de 32x32 texels. Diversos padrões foram criados e utilizados para estudar a performance das caches sob tipos de acesso diferentes. Os padrões que merecem especial destaque são o padrão aleatório, onde as cores dos texels são sorteadas de maneira aleatória e o padrão "tabuleiro de xadrez", uma sucessão de texels bancos intercalados com texels pretos. Estes dois padrões merecem especial atenção pois cada um revela um detalhe do comportamento das *caches* de maneira distinta: o padrão aleatório é o pior caso, onde cada acesso a textura dependente ocorre numa posição diferente e não vizinha das anteriores, de forma que os mecanismos de cache devem ser ineficazes, tanto L1 quanto L2. Com o padrão "tabuleiro de xadrez", estamos sempre acessando as mesmas duas posições da textura dependente, mas de maneira intercalada. Supõem-se que neste caso, apesar de a cache L1 ser ineficaz, a cache L2 deve estar contribuindo para uma melhora da performance em relação ao pior caso. As demais texturas são obtidas intercalando valores intermediários de cinza entre os texels brancos e pretos da textura "tabuleiro de xadrez". Na prática o efeito obtido é o de uma rotação de linhas (ou colunas). Observa-se que no caso de linhas perfeitamente horizontais ou colunas bem verticais, a performance é bem próxima do máximo, situação na qual os dois *caches* estão suprindo de maneira satisfatória o *pipeline*.

Para modelar o efeito do tipo de textura na performance da GPU, foram utilizadas a primeira e segunda freqüências de máxima amplitude retornadas por uma Transformada Rápida de Fourier discreta aplicada às texturas dependentes. Foram levadas em consideração quatro freqüências: a freqüência de maior amplitude medida na direção vertical (fy1), a freqüência de maior amplitude medida na direção horizontal (fx1) , a segunda freqüência de maior amplitude na direção vertical (fy2) e a segunda freqüência de maior amplitude na direção horizontal (fx2). Como as texturas tem dimensões de 32x32 *texels*, a freqüência máxima possível é 16, quando se tem uma oscilação completa a cada dois *texels*. A freqüência mínima é 0, quando não se tem oscilações.

Com base nesses parâmetros, alguns padrões foram estabelecidos, por exemplo: quando temos linhas verticais ou horizontais o desempenho das *caches* não é diminuído, ou seja, quando a freqüência de mais alta amplitude é zero, a performance é próxima do máximo. No caso de texturas aleatórias, onde a segunda e a primeira freqüência de mais alta amplitude são próximas da freqüência máxima possível, tanto a L1 quanto a L2 não estão sendo eficazes. Quando a segunda freqüência de mais alta amplitude é muito diferente da freqüência máxima possível (ou é zero), o modelo utiliza apenas a freqüência de mais alta amplitude; Este é o caso das texturas derivadas do "tabuleiro de xadrez". Observa-se que quanto mais próxima a fx1 estiver da freqüência máxima possível, tanto menos eficiente será a cache L1 – como exemplo a textura "tabuleiro de xadrez".

O procedimento para obtenção da inclinação das retas é o seguinte:

- Verificar se fx1 ou fy1 é zero. Isto é, temos listras verticais ou horizontais; em caso afirmativo, a inclinação da segunda e terceira retas é igual. Isso indica que os caches L1 e L2 estão conseguindo suprir as necessidades do pipeline.

- Se fx1 ou fy1 diferente de zero, então parametrizar a segunda reta segundo uma equação que toma como entrada fx1 e gera como saída valor candidato da inclinação dessa reta beta_2 .

- Se a segunda freqüência horizontal (fx2) é próxima da freqüência máxima, então a terceira reta é parametrizada como função dessa freqüência e o valor da sua inclinação beta_3 é dado por uma função de fx2; senão a inclinação da terceira reta assume um valor constante.

- Somar beta_2 e beta_3 para obtenção de novo beta_3 .

Depois de composto o valor para a inclinação das retas eles são então utilizadas para obter a desaceleração devido ao acessos a texturas dependentes. O cálculo do valor candidato da inclinação da segunda reta é dado por uma equação que relaciona o coeficiente da inclinação da reta (beta_2) com o valor de da maior freqüência horizontal da textura independente, fx1:

$$\text{beta}_2 = 6.07\text{e-}7 + 9.49\text{e-}8 * fx1 \tag{5.8}$$

onde beta_2 é o coeficiente da inclinação da segunda reta no gráfico de tamanho de textura versus performance. Os coeficientes da equação (5.9) são obtidos

ajustando-se a melhor reta sobre um gráfico de fx1 versus beta_2. A função que relaciona beta_3 com fx2 é uma função degrau:

$$\text{beta}_3 = \begin{cases} 1.30 \cdot 10^{-7}, fx2 \leq 14 \\ 1.18 \cdot 10^{-6}, fx2 > 14 \end{cases} \quad (5.9)$$

Calculado o erro relativo entre o simulador e o valor medido para cada dado, observa-se que o simulador consegue reportar resultados com um erro relativo não maior do que 10% (na maioria dos casos) neste estágio – figura 5.12 – em alguns casos bastante característicos, texturas que são potências de dois, o hardware aparenta ser otimizado. Como tais situações são poucas e bem definidas, pode-se eventualmente estender o simulador para incluir um ganho de performance percentual para os casos de texturas potências de dois, o que não está sendo feito.

5.1.6 Discussão sobre os experimentos de validação

O hardware estudado expõe um grupo de recursos que podem ser explorados na utilização da GPU. Estes recursos podem ser, por exemplo, o número de pipelines, a freqüência de operação do processador, o número de fragmentos por triângulos, a freqüência com que desvios de execução de código aparecem, os acessos a texturas, dentre outros. Para avaliar a utilização de tais recursos pelo hardware se propõe um modelo de estimativa de performance e a sua implementação através de um simulador.

O modelo de previsão de performance da GPU apresentado é composto de diversos estágios. Estes estágios podem ser agrupados em série ou em paralelo. Cada estágio procura aproximar o comportamento de performance de um recurso do hardware. A utilização de um recurso do hardware pode gerar uma perda na performance da GPU. O modelo proposto é um modelo de desaceleração, onde cada estágio introduz um fator de perda de performance após a performance máxima ser calculada. Para cada estágio do modelo, uma implementação foi feita para simular as características de performance do recurso correspondente. Para alguns recursos foi realizada uma série de testes que foram posteriormente comparados com os resultados obtidos via medição num hardware escolhido como sistema de referência.

Os parâmetros de entrada dos módulos foram variados sobre uma faixa de valores que se julgou adequada. Para acessos a texturas independentes, por exemplo, variou-se o tamanho da textura desde tamanhos de 2x2 texels até tamanhos de 4096x4096 texels – o tamanho máximo de texturas permitido pelo hardware de referência. O tamanho dos triângulos foi variado entre 3 e 960.000 fragmentos por triângulo. A carga sobre o processador de fragmentos foi também variada mudando o número de fragmentos a serem processados e o número de ciclos de cada programa a ser executado. Em todos os casos, procurou-se manter o erro relativo do simulador abaixo de 10%.

Em algumas situações é difícil manter o erro do simulador abaixo de 10% devido a grandes flutuações momentâneas do sistema sendo analisado. Assim como a GPU é composta por diversas partes, o sistema no qual ela está instalada também é composto de diversos subsistemas. Cada um destes subsistemas pode variar sua carga rapidamente em diversos momentos. Por exemplo, quando um acesso ao disco rígido do sistema ocorre, uma interrupção à CPU é disparada solicitando sua atenção. Em geral

estes picos de sobrecarga são difíceis de ser previstos mas têm uma duração curta. Os efeitos de oscilação causados pelos picos de sobrecarga do sistema podem ser minimizados se a carga sobre a GPU for razoavelmente elevada frente às outras demandas do sistema. Assim, os picos de oscilação ficam mascarados pelo trabalho que a GPU realiza. Por isso, sempre que possível tentou-se impor uma carga média a grande para a GPU durante os testes. Em geral, quanto maior a carga sobre a GPU, tanto maior a estabilidade dos valores medidos.

5.2 Verificação do Simulador com Uma Aplicação Complexa

Para verificação do simulador de performance, analisou-se a simulação de uma aplicação de tempo-real. A aplicação escolhida foi a implementação de um sistema genérico de massa-mola (DIETRICH, 2006). Um sistema massa-mola é composto por partículas (massas) ligadas umas às outras através de molas elásticas. Toda a simulação do sistema massa-mola é realizada na GPU. Este experimento foi realizado apenas na placa de vídeo GeForce 7800 da NVidia porque o hardware da ATI não suporta a extensão OpenGL GL_TEXTURE_RECTANGLE_NV necessária para implementação do programa.

5.2.1 Sistema Massa-Mola: Problema

O problema da simulação de um sistema massa-mola consiste na resolução da equação fundamental do movimento:

$$\mathrm{F} = m \cdot a \tag{5.11}$$

onde m é a massa de cada partícula P e a é a aceleração causada pela força F. As forças podem ser divididas em forças internas e externas. As forças internas resultam das tensões das molas ligando a partícula P_i com os seus vizinhos P_j:

$$\mathrm{F}(P) = \sum k_{ij} \cdot (l_{ij} - l_{0\mathrm{ij}}) \tag{5.12}$$

onde l_{ij} e $l_{0\mathrm{ij}}$ são respectivamente o comprimento atual e o comprimento de repouso da mola conectando a partícula P_i com a partícula P_j e k_{ij} é o coeficiente de rigidez da mola.

Não importando quantas molas e quantas partículas existam no sistema, a força aplicada a uma dada partícula deve ser transmitida a todas n molas às quais ela está conectada. As forças dependem apenas da compressão/dilatação da mola. Existem diversas partículas para atualizar, cada uma requerendo algumas operações aritméticas que são independentes dos cálculos das demais partículas. A implementação do modelo é separada na representação das partículas e das molas.

Os atributos de uma partícula são a posição, massa e velocidade. A posição (vetor de três dimensões) e massa (escalar) são armazenadas em uma textura de ponto flutuante de 32 bits por canal com quatro componentes (canais). Uma vez que no modelo todas as forças entre as partículas são posicionais, o cálculo da nova posição não depende da velocidade e portanto não é armazenada explicitamente. Os atributos de

uma mola são o comprimento de repouso e o coeficiente de rigidez. A mola representada não tem massa.

5.2.2 Sistema Massa-Mola: Simulador

O sistema massa-mola consiste de uma aplicação executada na CPU que utiliza a API OpenGL para executar diversos programas de fragmentos na GPU. A verificação do nosso simulador de performance se concentrou no programa de fragmentos responsável por calcular a soma das forças que atuam sobre uma determinada partícula. Na abordagem utilizada, a topologia é codificada em duas texturas (TEJADA, 2005). Uma textura armazena os vizinhos (textura de vizinhos) de cada partícula e a outra é utilizada como uma tabela de alocação de vizinhos (textura de vizinhança). A textura de vizinhança serve para armazenar um ponteiro para uma posição na textura de vizinhos, onde a informação sobre os n vizinhos de cada partícula está armazenada. Esta informação inclui um ponteiro para uma partícula (adjacente) e os parâmetros mecânicos da mola. O número de vizinhos (n) é armazenado no canal azul da textura de vizinhança. Neste problema, cada fragmento processado pelo processador de fragmentos representa uma partícula. O programa de fragmentos é capaz de recuperar as informações sobre todas as molas conectadas a cada partícula a partir da textura de vizinhança e da textura de vizinhos. O programa de fragmentos realiza quatro acessos a três texturas diferentes – figura 5.15. Essas texturas de 4 canais são todas armazenadas em precisão de 32 bits ponto flutuante por canal. As texturas de vizinhança e de geometria tem dimensões iguais: 1024x1024 texels. A textura de vizinhos tem dimensão de 512x256 texels.

Acompanhe na figura 5.15. Os fragmentos A, B, C, etc. são fragmentos vazios que vêm do processador de vértices com coordenadas de textura. O programa realiza então os acessos a texturas e as operações matemáticas necessárias. O resultado final (a soma das forças que atuam sobre uma partícula) é salvo no framebuffer, mas não é apresentado na tela, já que seu conteúdo não tem significado visual. A aplicação foi instrumentada imediatamente antes e imediatamente depois da execução do programa de fragmentos, para medir o tempo necessário para processar todos os fragmentos.

O programa de fragmentos avaliado – figura 5.16 – efetua quatro acessos a texturas. Os acessos 1 e 2 são independentes e são realizados fora do laço. Os acessos 3 e 4 são dependentes e são efetuados dentro do laço. O acesso 4 depende do acesso 3. O acesso 3 depende do acesso 2. A figura 5.15 é uma representação dos passos no código da figura 5.16.

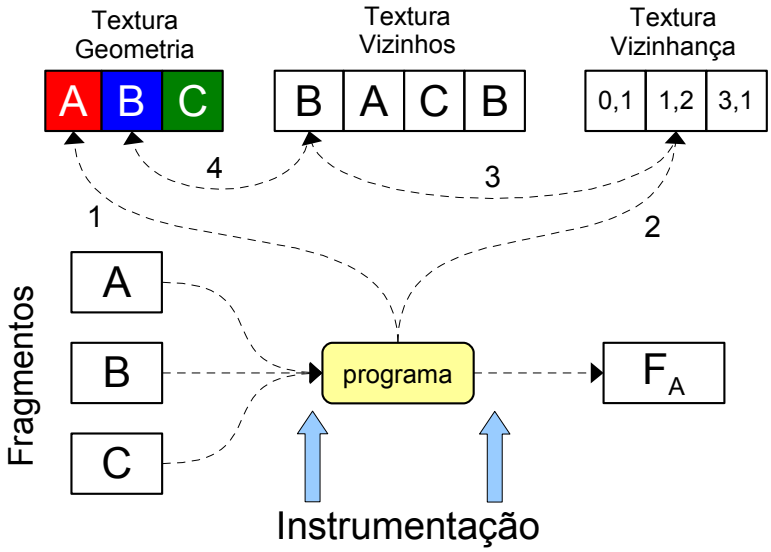

Figura 15: Programa de Fragmentos utilizado para verificação do simulador com a aplicação do sistema massa-mola sendo executado na GPU.

Com o objetivo de verificar os resultados do simulador frente à performance medida na GPU, efetuamos algumas mudanças na aplicação para simplificar a análise. Assim como na construção do simulador, onde construiu-se sucessivamente estágios que aproximavam cada vez mais a previsão de performance esperada, foram feitas diversas simplificações na aplicação escolhida. Essas simplificações foram sendo removidas gradualmente e acompanhou-se o desempenho do simulador. Para os experimentos de verificação, os acessos a textura foram modificados de três maneiras distintas:

- Experimento sem texturas: Todos os acessos a texturas no programa da figura 5.16 foram substituídos pela instrução aritmética sqrt (raiz quadrada). A instrução sqrt substitui o ciclo do programa que seria utilizado pelo acesso a textura. O resultado final do programa é fisicamente incorreto. Como não interessa a correção física da aplicação, esta situação serve para os propósitos de medição de tempo de execução.

- Experimento sem texturas dentro do laço: os acessos a texturas dentro do laço (acessos 3 e 4 na figura 5.16) foram substituídos por instruções aritméticas sqrt (raiz quadrada). O resultado final do programa de fragmentos é fisicamente incorreto, mas, novamente, para fins de comparação de performance com o simulador, serve aos propósitos. Os acessos a texturas independentes (acessos 1 e 2) não foram retirados.

- Experimento com todos acessos a texturas: todos os acessos a texturas permanecem no código como na sua concepção original. Dois acessos dependentes e dois acessos indepentendes são realizados. O acesso 4 depende do acesso 3. O acesso 3 depende do acesso 2. E os acesssos 1 e 2 são independentes. O resultado obtido com a execução deste código é fisicamente correto.

```
texld ...   // acesso a textura de geometria (1)
texld ...   // acesso a textura de vizinhança (2)

...         // instruções aritméticas

loop ...    // laço repetido 30 vezes

  ...       // instruções aritméticas
  texld ... // acesso a textura de vizinhos (3)
  texld ... // acesso a textura de geometria(4)
  ...       // instruções aritméticas
  break     // saída antecipada do laço

endloop     // fim do laço

mov oC0, r0
```

Figura 16: Pseudo código executado pela aplicação massa-mola; as instruções aritméticas não estão representadas.

Para verificar o funcionamento do simulador com relação à aplicação escolhida e com relação à presença de instruções de desvio, o programa de fragmentos foi compilado para três perfis de linguagem assembly:

- arbfp1 – neste perfil desvios de código não são permitidos. Ou seja, os laços e desvios condicionais são removidos do programa. Os laços são desenrolados pelo compilador e os desvios condicionais eliminados.

- arbfp40 – este perfil de programas de fragmento aceita desvios de código. Laços no programa original são mantidos e não são desenrolados. Desvios de código condicionais também são permitidos, mas instruções de finalização prematura de laços não foram utilizadas. O laço uma vez iniciado, deve executar todas as iterações previstas.

- arbfp40 com interrupção de laço – este é o mesmo perfil que o anterior, porém foi utilizada a instrução “break” que faz com que um laço seja interrompido prematuramente.

Nove combinações são possíveis entre os perfis para o qual o programa de fragmentos foi compilado e a variação nos acessos as texturas. A tabela apresenta as possíveis combinações entre os perfis de linguagem assembly e a variação nos acessos a texturas. Cada linha da tabela tem um tipo de acesso a texturas e cada coluna um perfil de linguagem assembly:

Tabela 5.1: Performance medida e simulada para o modelo massa-mola

		Desenrolado	Laço	Laço+Parada
Sem Textura	Medido	**344**	**206**	**295**
	Simulado	*338*	*209*	*307*
	Erro Relativo	0,01	0,01	0,04
Sem Textura no Laço	Medido	**171**	**122**	**270**
	Simulado	*172*	*123*	*253*
	Erro Relativo	0,01	0,01	0,06
Com Textura	Medido	**67**	**63**	**162**
	Simulado	*68*	*59*	*177*
	Erro Relativo	0,01	0,06	0,09

Foram realizadas 9 medidas diferentes utilizando o sistema massa-mola. Variando a estrutura do programa (com e sem desvios) e o padrão de acesso a texturas (com e sem acessos, ver tabela 5.1). Todas as medidas foram realizadas utilizando sempre o mesmo número de fragmentos (131.072). O número de triângulos (dois) também foi mantido constante. Os valores medidos e simulados são apresentados em FPS.

Para a estimativa da performance, utiliza-se o simulador com os estágios configurados de acordo com a arquitetura sendo utilizada e com a aplicação que está sendo analisada. As características da arquitetura, tais como número de pipelines e freqüência de operação são conhecidas e não mudam. Da aplicação obtém-se o número de VLIWs que compõem o programa de fragmentos (através do aplicativo NVShaderPerf) e o número de fragmentos a serem processados. Para os acessos a texturas dependentes, uma análise das suas freqüências mostrou que elas se enquadram no caso de linhas verticais (figura 5.14), que praticamente não apresentam impacto na performance devido ao seu padrão de acesso. Os tamanhos e formatos das texturas também não impuseram nenhuma degradação na performance.

Pode-se observar da tabela 5.1 que o erro relativo aumenta de cima para baixo e da esquerda para direita. Isso se deve ao fato de que no topo à esquerda, apenas o primeiro estágio do simulador é utilizado. Neste caso, o erro relativo deve-se unicamente a este estágio. Quando desvios de código incondicionais (laços) ou incondicionais (interrupções) estão envolvidos, o segundo estágio do simulador deve ser utilizado. Utilizando dois estágios do simulador, o erro relativo precisa ser composto e é aceitável, nesta situação, um erro de até 22% (de acordo com a regra de composição do erro apresentada anteriormente). Quanto acessos a texturas são utilizados conjuntamente com desvios, todos os estágios do simulador estão sendo utilizados e, portanto, o erro deve ser composto levando em conta o erro parcial de cada estágio. Como mencionado anteriormente, se cada estágio permite um erro de até 10%, é razoável esperar um erro composto de até 33%.

5.3 Resultados da Validação e da Verificação

O simulador de características de performance da GPU apresentado neste trabalho compõe-se de estágios que podem ser combinados de maneira seqüencial ou paralela. O primeiro destes estágios, é responsável pela previsão de uma performance

máxima possível baseada em parâmetros de desempenho da GPU e da carga de trabalho aplicada. Os estágios seguintes são destinados a introduzir uma redução de performance devido a utilização de recursos da GPU. Estes estágios foram concebidos para abranger algumas situações que se julgaram adequadas e significativas para a simulação de performance de uma GPU.

Cada um dos estágios apresenta um erro percentual médio que se tentou manter sempre abaixo da faixa de 10% em relação a medidas realizadas no hardware de referência. Associação em série dos estágios do simulador pode levar a erros de até 30%. Durante os estudos das características de performance do hardware percebeu-se que há otimizações para algumas situações. Por exemplo, quando a textura utilizada tem dimensões (em texels) que não são potências de dois, a performance sofre uma degradação de 20% a 30% algumas vezes. Com a verificação do simulador frente a uma aplicação real, percebe-se que o simulador consegue cobrir a maioria das possíveis situações de utilização da GPU para processamento genérico.

6 Conclusões

Este trabalho teve como objetivo principal a modelagem do hardware gráfico e como melhorar sua performance. Para isto foi necessário entender o funcionamento de cada uma das partes que compõe a GPU e alguns relacionamentos entre elas. Este modelo teve a sua implementação representada por um simulador de estimativa de performance da GPU. O simulador focou-se no processador de fragmentos da GPU, assumindo que os demais componentes da GPU não são gargalos para o sistema. Assim como o modelo, este simulador é constituído de diversos estágios que são utilizados para obtenção da estimativa de performance. O estágio inicial é responsável por uma estimativa de performance inicial; os estágios seguintes são responsáveis pela introdução de fatores que podem aproximar melhor a performance esperada. O simulador foi validado utilizando-se medidas sintéticas e posteriormente verificado comparando-se seus resultados com uma aplicação real. O simulador, no atual estado de desenvolvimento, consegue aproximar algumas situações possíveis de utilização da GPU, e é expansível para incluir mais detalhes sobre a GPU posteriormente.

O modelo proposto pretende aproximar o comportamento do hardware gráfico com parâmetros de alto nível, sem tentar descrever em detalhes os componentes da GPU. O modelo tenta explicar a performance do hardware gráfico utilizando como parâmetros algumas medidas de recursos acessíveis pelo usuário, tais como número de pipelines presente na GPU, freqüência de operação do processador gráfico, tamanho das texturas utilizadas, tamanho dos triângulos usados para compor a cena, dentre outros. Quanto maior o número de parâmetros utilizados pelo simulador, maior o nível de detalhamento alcançado pelo simulador. Como o simulador não utiliza como entrada todos os parâmetros do hardware e da aplicação possíveis, seu alcance é limitado.

O simulador, conforme implementado, não consegue aproximar todos os relacionamentos existentes entre os componentes de uma GPU. Apesar do modelo tentar manter a maior abstração possível do hardware, algumas relações entre seus componentes são necessárias para que seja possível descrever corretamente o comportamento do sistema sendo modelado. Como nem todas as relações entre os componentes da GPU estão presentes, em algum nível de detalhamento o simulador pode deixar de fornecer resultados adequados. Como este simulador é expansível através da utilização de mais estágios (ou substituindo os atuais), o seu grau de detalhamento pode ser refinado gradualmente.

O simulador foi construído com uma meta de erro relativo máximo de 10% em cada estágio. Com a composição do erro relativo dos 3 estágios do simulador tolera-se um erro total de até 33%. O simulador reporta valores corretos quando utilizado em situações sintéticas, mas é importante lembrar que nestas situações apenas um parâmetro está sendo variado em cada experimento. Em problemas não-sintético, em

geral, os parâmetro do sistema não estão sob controle sendo permitido a mais de um deles variar simultaneamente. No caso da aplicação de simulação de sistemas massa-mola – seção 5.2 – o erro atingido chega a valores em torno de 10% por estágio. É interessante notar que quando acessos a texturas não estão sendo realizados, o erro relativo fica abaixo de 2%. Quando laços e desvios condicionais são utilizados (sem acessos a texturas), o erro do simulador manteve-se abaixo do limite de 10% por estágio. Em algumas situações um erro maior pode ser produzido por rápidas flutuações da carga em outras partes do sistema sendo analisado, tais como o subsistema de entrada/saída ou sobre a CPU. Estas flutuações são responsáveis por erro não sistemático que pode aparecer em uma medida, mas não se reproduzir na próxima. Erros sistemáticos podem ser introduzidos devido a falta do correto relacionamento entre os dados de entrada do simulador, ou até mesmo a falta destes dados de entrada.

O modelo apresentado neste trabalho foi construído tendo em vista a capacidade de expansão. Um simulador implementado, segundo o modelo, é composto por estágios que podem ser combinados, alterados ou substituídos. Os estágios que compõem o simulador ainda não são capazes de imitar o comportamento da GPU em todas as situações possíveis. Para abranger todos os regimes de operação que uma GPU pode suportar, outros estágios devem ser construídos. Estágios que simulem o acesso a texturas dependentes com mais de um nível de dependências precisam ser implementados. Atualmente, o simulador não conta com um estágio para simular programas que tenham mais de um nível de blocos de código com instruções de controle de fluxo. O simulador também não leva em conta efeitos de latência entre os estágios do pipeline, exceto para os acessos a texturas, onde este dado já está incluído no tempo de resposta da unidade de leitura de texturas. Seria desejável que o simulador tivesse algum conhecimento sobre os tempos necessários para escritas em texturas. Não se sabe se estas escritas são ou não realizadas através das memórias cache ou se para isso é necessário que o módulo de texturas acesse diretamente a memória de vídeo, por exemplo. O simulador também não leva em conta a perda de performance quando o número de registradores disponíveis é excedido. Atualmente o hardware suporta a utilização de um número limitado de registradores de uso geral por fragmento sem a perda de performance. A partir deste ponto a performance é diminuída por não haver registradores para acomodar todos os dados necessários ao processamento de todos os fragmentos.

Além de novos estágios que podem ser construídos, os estágios atuais também podem ser estendidos para incluir novas características (parâmetros). Estas novas características podem permitir a um dado estágio simular situações antes não abrangidas ou podem diminuir a incerteza do simulador. A inclusão destes novos parâmetros agrega maior detalhamento ao simulador. Com uma maior detalhamento do hardware, novos relacionamentos entre as suas diversas partes podem ser reproduzidos e inseridos no modelo. O simulador pode responder com maior precisão conhecendo mais relacionamentos existentes entre os componentes do hardware sendo simulado. Como exemplo podemos citar uma estimativa mais precisa para o tamanho das memórias cache de texturas. Outros estágios podem até mesmo simular os demais componentes da GPU. Os demais estágios do pipeline gráfico discutidos no capítulo 2 deste trabalho também são passíveis de serem simulados.

Utilizando a abordagem de estágios, mais componentes do sistema podem ser representados no simulador, tais como a comunicação entre a CPU e a GPU ou até mesmo a comunicação entre duas GPUs. Os modelos mais recentes de GPUs

apresentam a capacidade de trabalhar coordenadamente para divisão de tarefas. A NVidia chama essa associação entre GPUs de SLI e a ATI implementa tecnologia semelhante sob o nome de Cross-Fire. A versão 10 da API DirectX impõe novas características ao hardware de processamento gráfico. Estas novas características também são interessantes de serem simuladas.

Referências

ABEL, J. et al. Applications Tuning for Streaming SIMD Extensions, **Intel Technology Journal Q2**, [S.l.] 1999.

ATI CORPORATION: ***Radeon X1900 Series.*** Disponível em: <http://www.ati.com/products/RadeonX1900/index.html>. Acesso em 29 maio 2006.

BAUMAN, D. G70 NVIDIA GeForce 7800 GTX Review. Disponível em: <http://www.beyond3d.com/previews/nvidia/g70/>. Acesso 26 maio 2006.

BLAS, A. ; HUGHEY, R. Explicit SIMD Programming for Asynchronous Applications. In: IEEE INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS 2000. **Proceedings...** [S.l.]: IEEE, 2000. p. 258-267.

BOUATOUCH K.: *Overview of Parallel Graphics Hardware*, **Pratical Parallel Rendering**, A K Peters, 2002.

BRATLEY, P.; FOX, B. L.; SCHRAGE L. E. **A Guide to Simulation.** 2nd ed. New York: Springer-Verlag, 1987.

BUCK, I.; FATAHALIAN, K.; HANRAHAN, P. GPUBench: Evaluating GPU performance for numerical and scientific applications. In: ACM WORKSHOP ON GENERAL-PURPOSE COMPUTING ON GRAPHICS PROCESSORS 2004. **Proceedings...** New York: ACM, 2004.

DESIKAN, R.; BURGER, D.; KECKLER, S. W. Measuring Experimental Error in Microprocessor Simulation. **ACM SIGPLAN Software Engineering Notes, New York**, r. 26, n. 3, p. 266-277, 2001.

DIRECTX DEVELOPER CENTER: <http://msdn.microsoft.com/directx/>. Acesso 29 maio 2006.

DYM, C. L.; IVEY, E. S.; **Principles of Mathematical Modeling.** New York: Academic Press, 1980.

FAHRINGER, T.: Evaluation of Benchmark Performance Estimation for Parallel Fortran Programs on Massively Parallel SIMD and MIMD Computers. In: EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING, 2. 1994. **Proceedings...** [S.l.]: IEEE, 1994. p. 449-456

FATAHALIAN, K.; SUGERMAN, J.; HANRAHAM P. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In: ACM SIGGRAPH/EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE. **Proceedings...** New York: ACM Press, 2004. p. 133-137.

FISHER, J. A.; FREUDENBERGER, S. M. **Predicting Conditional Branch Directions From Previous Runs of a Program**. New York: ACM Press, 1992.

FLYNN, M. J.; Very high-speed computing systems. **Proceedings of the IEEE**, Piscataway, v. 54, n. 12, Dec. 1966.

GIBSON, J.; et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. **ACM SIGPLAN Notices**, New York, v. 35, n. 11, p. 49-58, 2000.

Gnuplot Homepage. Disponível em: <http://www.gnuplot.info/>, acesso em 29 maio 2006

GOODNIGHT, N. et al. A multigrid solver for boundary value problems using programmable graphics hardware. In: ACM SIGGRAPH/ EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE. **Proceedings...** New York: ACM Press, 2003. p. 102-111

GRIMM, S. et al. **Parallel peeling of curvilinear grids**. Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2004. (TR 186-2-04-07)

HALL, J.; CARR, N.; HART, J. **Cache and Bandwidth Aware Matrix Multiplication on the GPU**. Illinois, USA: University of Illinois, 2003. (Techinal Report UIUCDCS-R20032328)

HARRELL, C. B.; FOULADI, F. Graphics Rendering Architecture for a High Performance Desktop Workstation. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 20, 1993. **Proceedings...** New York: ACM Press, 1993. p. 93-100.

JACOBY, S. L. S.; KOWALIK, J. S. **Mathematical Modeling With Computers**. Englewood Cliff: Prentice-Hall, 1980.

JAIN, R. **The Art of Computer Systems Performance Analysis.** New York: John Wiley & Sons, 1991.

KILGARIFF, E.; FERNANDO, R. The GeForce 6 Series GPU Architecture. **GPU Gems 2.** Boston, USA: Addison-Wesley, 2005.

LILJA, D. J. **Measuring Computer Performance.** Cambridge: Cambridge University Press, 2000.

LINDHOLM, E.; KILGARD, M. J.; MORETON, H. A User-Programmable Vertex Engine. In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 28, 2001. **Proceedings...** New York: ACM Press, 2001. p. 149-158.

MARK, W. R.; PROUDFOOT, K. Compiling to a VLIW Fragment Pipeline. In: ACM SIGGRAPH/ EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE. **Proceedings...** New York: ACM Press, 2001. p. 47-56.

NICHOLS, M. A.; SIEGEL, H. J.; DIETZ, H. G. Data Management and Control-Flow Aspects of an SIMD/SPMD Parallel Language Compiler. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v. 4, n. 2, p. 222-234, 1993.

NVIDIA CORPORATION: **GeForce 7 Series**. 2005. Disponível em: <http://www.nvidia.com/page/geforce7.html>. Acesso em: 29 maio 2006.

NVIDIA CORPORATION: **Nvshaderperf - nvidia shader scheduling analysis**. 2006. Disponível em: <http://developer.nvidia.com/object/nvshaderperf_home.html>. Acesso em: 3 maio 2006.

OLK, J. G. E. et al. A Programming and Simulation Model of a SIMD-MIMD Architecture for Image Processing. In: WORKSHOP ON COMPUTER ARCHITECTURE FOR MACHINE PERCEPTION. **Proceedings...** [S.l.]: 1995. p. 95-108.

PATTERSON, D. A.; HENNESY J. L. **Computer Architecture: A Quantitative Aproach.** 2nd ed. San Francisco: Morgan Kaufmann, 1996.

PUERCELL, T. J. et al. Ray tracing on programmable graphics hardware. **ACM Transactions on Graphics**, New York, v. 21, n. 3, p. 703-712, July 2002.

SÁNCHEZ, J.; GONZÁLEZ, A. Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture. In: ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 33, 2000. **Proceedings...** New York: ACM Press, 2000. p. 124-133.

TEJADA, E.; ERTL T. Large Steps in GPU-based Deformable Bodies Simulation. **Simulation Pratice and Theory**, v. 13, n. 8, p. 703-715, 2005.

TOMBOULIAN, S. Overview and Extensions of a System for Routig Directed Graphs on SIMD Architectures. In: FRONTIERS OF MASSIVELY PARALLEL COMPUTATION, 2, 1988. **Proceedings...** [S.l.]: 1988, p. 63-67.

TORBORG, J. G. A Parallel Processor Architecture for Graphics Arithmetic Operations, In: ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 14, 1987. **Proceedings...** New York: ACM Press, 1987. p. 197-204.

TRANCOSO, P.; CHARALAMBOUS M. Exploring graphics processor performance for general purpose applications. In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN, 8, 2005. **Proceedings...** [S.l.]: 2005. p. 306-313.

TRIOLET D. : **The Radeon X1900 XTX, X1900 XT and X1900 CrossFire in tests.** Disponível em: <http://www.behardware.com/articles/605-1/the-radeon-x1900-xtx-x1900-xt-and-x1900-crossfire-in-tests.html>. Acesso em: 26 maio 2006.

WILSEY, P. A.; HENSGEN, D. A.; ABU-GHAZALEH, N. B. The Concurrent Execution of Non-communicating Programs on SIMD Processors. In: FRONTIERS OF MASSIVELY PARALLEL COMPUTATION, 4, 1992. **Proceedings...** [S.l.]: 1992, p. 29-36.

YE, Z. A.; SHENOY, N.; PRITHVIRAJ, B.: **A C Compiler for a Processor with a Reconfigurable Functional Unit.** New York: ACM Press, 2000.