2
Programação Genérica em Placa Gráfica

A Unidade de Processamento Gráfico – do inglês “Graphics Processing Unit”(GPU) foi desenvolvida inicialmente como um hardware destinado a aumentar a eficiência e o poder de processamento gráfico para tarefas de renderização. Hoje, a GPU apresenta-se como um hardware de processamento versátil e de alto poder de computação. Tornou-se uma possibilidade real na busca por soluções para processamento em grandes volumes de dados, seja como complemento, seja como alternativa ao uso de CPUs multicore, ou de sistemas distribuídos.

Existem três fatores que influenciaram o rápido desenvolvimento do hardware gráfico. Em primeiro lugar aponta-se o crescimento do poder de processamento das GPUs comparado ao das CPUs (ver gráfico na Figura 2.1). Em segundo lugar, o fato de que tal performance esteja disponível a custos relativos baixos. O aumento da programabilidade das GPUs completa o trio de motivos para a sua enorme aceitação, favorecido pela introdução de linguagens de programação de shaders, como solução de programação alto nível, em contrapartida à programação em assembly.

Em relação ao desenvolvimento e aceitação do hardware gráfico especificamente para programação de propósito geral, destacam-se ainda o suporte à computação em ponto flutuante 32-bits IEEE, com precisão suficiente para diversas áreas de computação científica e o recente impulso causado pela introdução das linguagens de computação genérica em GPU, que não requerem o conhecimento gráfico a priori para o seu aprendizado (CUDA08, CTM08).

As aplicações mais beneficiadas pela arquitetura da GPU possuem três características principais em comum (Owe08). Demandam grandes volumes de recursos computacionais normalmente relacionados a computações sobre volumes de dados massivos. Apresentam um domínio amigável a formulações paralelas. E, como terceira característica, valorizam produtividade (do termo inglês, “throughput”, quantidade de informação processada em um intervalo de tempo), mais do que latência (tempo de reação entre um estímulo e uma resposta a ele), influenciadas pelo fato de que operações em processadores modernos demandam intervalos de ordens de grandeza de nanosegundos para

PUC-Rio - Certificação Digital Nº 0510966/CA

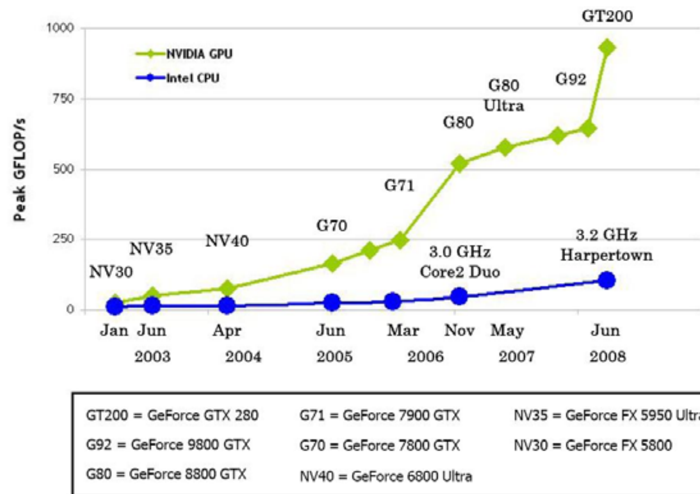

Figura 2.1: Comparação GFLOP/s CPU $\times$ GPU (CUDA08)

serem computadas.

Neste capítulo, apresentamos o contexto inicial para o qual o hardware gráfico foi desenvolvido, as propostas de sua utilização para computação de propósito geral, padrões de programação paralela e uma comparação das arquiteturas da GPU e CPU modernas.

2.1 Pipeline de Processamento Gráfico

Nesta seção apresentamos uma visão geral sobre sistemas gráficos 3D. Em uma formulação geral, tais sistemas possuem a tarefa de gerar imagens que representem a visualização de uma cena virtual. Uma cena é definida pela geometria, orientação e propriedades dos materiais e de fontes de luz, enquanto a visualização da cena é descrita pela localização de uma câmera virtual.

Existe um conjunto de operações, executadas em uma determinada ordem, a serem aplicadas ao modelo da cena para sua visualização. É comum às APIs [1] de sistemas gráficos de tempo real como o DirectX3D e o OpenGL, organizar essas operações na forma de um pipeline, chamado *Pipeline de Processamento Gráfico*.

Em arquiteturas sem pipeline, durante a execução de um ciclo de instrução os recursos não utilizados no módulo ativo corrente ficam ociosos. Um pipeline consiste de diversas etapas encadeadas que podem ser executadas em paralelo. Arquitetura de pipeline é uma técnica utilizada para acelerar a velocidade de operação pelo aumento da quantidade de dados processados em um determinado intervalo de tempo. Para tanto, distribui os recursos de pro-

[1] do inglês, "Application Programming Interface", Interfaces de Programação de Aplicativos.

PUC-Rio - Certificação Digital Nº 0510966/CA

cessamento de forma a fazer os seus estágios trabalharem em paralelo e assim aumentar a produtividade, ou seja, aumentar o número de instruções executadas em uma unidade de tempo.

Para o contexto desta tese, o propósito de apresentar o *Pipeline de Processamento Gráfico* é observar o fluxo de dados e as transformações a que é submetido. Descreve-se, a seguir, uma visão geral de suas etapas, permitindo a posterior compreensão das formulações apresentadas de processamento genérico que utilizam o pipeline gráfico.

Durante a renderização, quando os modelos da cena são transformados para a geração de uma visualização, os dados são processados em quatro entidades principais: vértices, primitivas, fragmentos e pixels. Os dados são passados entre os estágios do pipeline na forma de fluxos de dados ("data stream") contendo as informações apropriadas a cada entidade. Uma abstração do pipeline real, com o foco no fluxo de dados, é apresentada na Figura 2.2, e seus estágios são descritos como (Fat08):

PUC-Rio - Certificação Digital Nº 0510966/CA

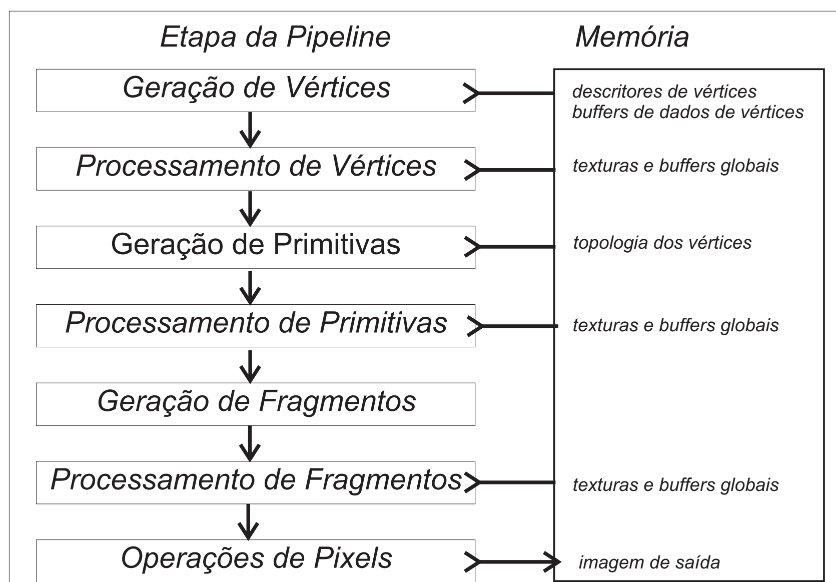

Figura 2.2: Abstração do Pipeline Gráfico

– *Geração de vértices*: Inicialmente, as APIs gráficas representam as superfícies como coleções de geometrias simples, tais como pontos, linhas, triângulos (mais usada) e polígonos. Cada primitiva é definida por um conjunto de vértices. Para iniciar a renderização, a aplicação passa para essa primeira etapa do pipeline uma lista de descritores de vértices. A partir dessa lista é construído um fluxo ("stream") de registros contendo os dados relativos aos vértices para o processamento subsequente, que são carregados da memória. Cada registro contém a posição 3D do vértice na

cena, além de parâmetros associados aos vértices definidos pela aplicação (exemplo: cor, normal, coordenadas de textura, entre outros).

– *Processamento de Vértices*: Esta etapa opera sobre o fluxo de vértices, executando um processamento por vértice de forma independente dos demais vértices.

  Para cada registro de entrada contendo a informação sobre um vértice, produz exatamente um registro de dados de saída contendo as novas informações do vértice processado. No contexto de renderização, a operação mais característica dessa etapa é a transformação individual de cada vértice pela projeção de sua posição 3D para o espaço 2D da tela. Outras operações realizadas nessa etapa são o cálculo de iluminação pela computação da interação do vértice com as luzes da cena, a transformação e a normalização de normais, a geração e transformação de coordenadas de textura, e a aplicação de cor de materiais.

– *Geração de Primitivas*: Essa etapa utiliza a descrição topológica dos modelos fornecida pela aplicação para agrupar conjuntos de vértices recebidos da etapa anterior em um fluxo ("stream") ordenado de primitivas. Cada registro de primitiva contém o agrupamento de um ou mais registros de vértices, de acordo com o tipo de primitiva (um vértice para primitiva ponto, dois para linha, três para triângulo, e assim por diante). A topologia também define a ordem das primitivas no fluxo de dados gerado.

– *Processamento de Primitivas*: Nessa etapa, cada primitiva de entrada é processada de forma independente e pode vir a gerar zero ou mais primitivas na saída. A saída gerada é um novo fluxo ("stream") de primitivas.

– *Geração de Fragmentos*: Nesta etapa é feita a rasterização, ou seja, a amostragem de cada primitiva para o espaço da tela. A amostragem determina as posições de pixels no espaço da tela cobertos por cada primitiva. A saída gerada é o fluxo das amostras criadas, onde cada amostra é representada por um registro denominado fragmento. As informações sobre um fragmento descrevem sua posição no espaço da imagem, sua distância à câmera virtual (coordenada z) e valores calculados como interpolações dos atributos associados originalmente aos vértices da primitiva que gerou a amostra.

– *Processamento de Fragmentos*: Esta etapa opera sobre o fluxo de fragmentos, processando cada fragmento de forma independente dos demais.

PUC-Rio - Certificação Digital Nº 0510966/CA

É responsável por descartar fragmentos ou determinar sua cor e opacidade. No contexto de renderização, tal coloração reflete o cálculo da interação da iluminação com as propriedades cromáticas e de materiais da superfície amostrada no fragmento. É comum que nessa etapa sejam feitos acessos com filtragem a memórias globais na forma de texturas 1D, 2D ou 3D.

– *Operações de Pixels*: Nessa etapa é calculada a contribuição de cada fragmento para o valor do pixel da imagem de saída que possui a mesma posição do fragmento.

  No contexto de renderização essa etapa é responsável por fazer o descarte de fragmentos ocultos por outras superfícies mais próximas a câmera (comparação da coordenada z do fragmento com uso do z-buffer), além da combinação de fragmentos amostrados de superfícies semi-transparentes (combinação ponderada pelo valor do canal alfa dos fragmentos).

Os pipelines gráficos de renderização originais do OpenGL e do DirectX3D são chamados pipelines gráficos de funcionalidade fixa por terem seu comportamento pré-definidos nas especificações dessas APIs (Ros06) e ocultarem do programador a implementação das operações em hardware.

PUC-Rio - Certificação Digital Nº 0510966/CA

2.1.1 Pipeline Gráfico Programável

A abstração do pipeline gráfico apresentada possui três etapas cujos comportamentos podem ser programados nas GPUs modernas pelo desenvolvedor da aplicação. São programáveis as etapas anteriormente denominadas Processamento de Vértices, Processamento de Primitivas e Processamento de Fragmentos. O comportamento dessas etapas pode ser modificado utilizando programas de shaders.

Os shaders são programas cujos conjuntos de instruções permitem definir as operações a serem realizadas pela GPU em etapas programáveis do pipeline gráfico. As operações descritas em um programa de shader substituem o comportamento padrão esperado para uma determinada etapa programável do pipeline gráfico, mais especificamente, as etapas programáveis são tratadas individualmente por programas de shader distintos que passam a ser inteiramente responsáveis por seu comportamento.

Com a introdução de GPUs programáveis, as principais APIs gráficas do mercado (DirectX3D e Opengl) passaram a incluir comandos relacionados à programação de shaders. Na época da introdução das placas programáveis, foram criadas diversas máquinas virtuais de shaders correspondendo a pro-

cessadores gráficos específicos disponíveis no mercado de hardware gráfico e associadas a linguagens de assembly próprias. Nesse modelo, o desenvolvedor de shader escreve seu programa na linguagem de assembly apropriada, o qual é passado da aplicação para a API gráfica escolhida que, então, devolve a representação binária de tal código. A representação binária é, por sua vez, passada ao hardware gráfico utilizando comandos da API escolhida para ser executada na GPU.

A introdução das linguagens de shaders [2] para programação de placas gráficas, tais como a linguagem CG, da NVidia, a GLSL, do consórcio OpenGL e a HLSL da Microsoft, passou a fornecer um nível mais alto de abstração para a descrição dos shaders, facilitando sua expressão, leitura e reutilização pelos desenvolvedores de shaders. Nesse modelo, o código de shader é transformado, pelo compilador da linguagem, em "bytecode", o qual por sua vez é transformado em binário específico para cada placa gráfica pelo driver gráfico em tempo de execução.

As linguagens de shader oferecem suporte a diversos tipos de dados e de instruções de controle de fluxo, mas não contêm primitivas relacionadas explicitamente à execução paralela. O código de um shader descreve uma função a ser aplicada a um único registro de uma entidade, cujo tipo é determinado pelo estágio do pipeline correspondente ao shader (vértice, primitiva ou fragmento). O processamento paralelo sobre os fluxos de dados das entidades acontece de forma transparente, replicando as rotinas de processamento de forma distribuída nos processadores da placa.

Cada execução de um shader sobre um registro pode ser abstraída como uma chamada de função que executa uma sequência de operações de maneira completamente isolada do processamento dos demais registros no fluxo da entidade correspondente. O modelo de programação das linguagens de shaders não permite repassar dados gerados por um elemento para o processamento de outro registro do mesmo nível do pipeline.

Em geral, cada etapa programável recebe um número limitado de vetores 4D de ponto flutuante com 32 bits, pode utilizar registros de leitura e escrita por elemento processado, além de registros constantes, inicializados antes da execução do shader e residente entre as etapas. As quantidades disponíveis de cada um desses tipos de memória estão sendo aumentadas com a evolução das arquiteturas. Além dos registros de dados dos fluxos de entrada e saída, os shaders podem acessar, sem modificar, grandes buffers globais de dados.

[2] A origem das linguagens de shaders é anterior às GPUs programáveis e está relacionada a geração de efeitos sofisticados na renderização de cenas 3D em tempos de processamento não interativos, sendo uma das primeiras conhecidas a "RenderMan Shading Language"(1980) (Fer03).

PUC-Rio - Certificação Digital Nº 0510966/CA

Antes da execução, esses buffers são inicializados pela aplicação para conter informações de parâmetros e texturas dos shaders.

Os shaders responsáveis por alterar o comportamento das etapas programáveis de processamento de vértices, primitivas, e fragmentos são denominados, respectivamente, "vertex shader", "geometry shader"e "fragment shader"[3]. Cabe ressaltar que a implementação de um shader para uma determinada etapa programável do pipeline não implica a obrigatoriedade de se implementar as outras duas, as quais, não sendo substituídas por shaders, continuam a exercer o comportamento padrão do pipeline fixo.

É interessante perceber que as três etapas hoje programáveis executam processamento de uma entidade para produzir o mesmo tipo de entidade (vértice para vértice, primitiva para primitiva e fragmento para fragmento), enquanto as etapas mantidas como fixas executam transformações no tipo da entidade de entrada para um novo tipo de entrada de saída (Owe07).

A seguir, os três tipos de shader são apresentados com o foco no fluxo de dados.

PUC-Rio - Certificação Digital Nº 0510966/CA

O "vertex shader"

Um vertex shader pode ser escrito contendo todas as funcionalidades originalmente associadas a essa etapa pelo pipeline fixo ou nenhuma delas, mas não é possível escrever um vertex shader para algumas das funcionalidades originais e usar a versão da funcionalidade fixa para as demais.

Para executar as transformações em um vértice, o "vertex shader"tem acesso a informações de entrada como sua posição, cor, coordenadas de textura e atributos associados aos vértices pela aplicação (ver Figura 2.3). O código de vertex shaders pode conter qualquer sequência de operações entre o conjunto de instruções permitidas com a obrigação única de gerar os dados de posição para o vértice de saída. Atributos como informações de cor e coordenada de textura são opcionais, mas o vertex shader possui a capacidade de emitir como saída um número limitado de vetores 4D de dados de ponto flutuante de 32 bits que serão interpolados pelo rasterizador. Placas recentes incluíram a capacidade de fazer acesso a textura nessa etapa.

O "geometry shader"

Esse estágio foi recentemente exposto como programável (a partir da série 8.0 da NVIDIA (GeForce 8800 Technical Brief)). Um programa de "geometry shader"descreve computações sobre as primitivas geométricas, onde

[3]Também denominados na literatura "vertex program", "geometry program"e "fragment program".

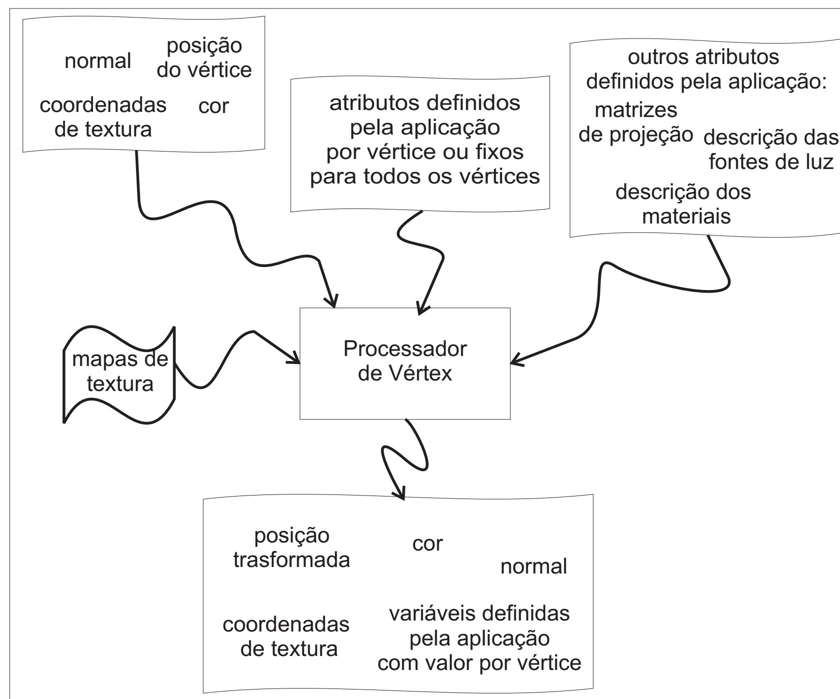

Figura 2.3: Entrada e saída de dados no processamento de vértices

PUC-Rio - Certificação Digital Nº 0510966/CA

cada primitiva é trabalhada isoladamente, podendo gerar novas primitivas gráficas como pontos ("point list"), linhas ("line strips") e triângulos ("triangle strips"), a partir das enviadas originalmente para o pipeline.

Um programa de "geometry shader"recebe como entrada acesso a uma primitiva inteira, ou seja, tantos vértices quantos corresponderem ao tipo da primitiva, podendo conter informações de adjacências (ver Figura 2.4).

Como saída, repassa para o estágio de rasterização zero ou mais primitivas de um mesmo tipo ("point list", "line strips"ou "triangle strips"). A introdução do "geometry shader"possibilita a execução em GPU de modificações na malha dos modelos da cena, antes possíveis apenas em CPU.

O "fragment shader"

Para executar as transformações em um fragmento, o "fragment shader"tem acesso a informações de entrada como a posição do fragmento, coordenadas de textura, além de valores interpolados durante a rasterização (ver Figura 2.5). O "fragment shader"não pode modificar as coordenadas de um fragmento, mas pode computar informações de cor e profundidade. Como saída, pode gerar um conjunto de vetores 4D de ponto flutuante em 32 bits que normalmente representam cores.

O "fragment shader"pode acessar a memória de uma ou mais texturas inúmeras vezes de diferentes formas e combinar os valores lidos como necessário. O resultado de uma leitura de textura pode ser usado como base para

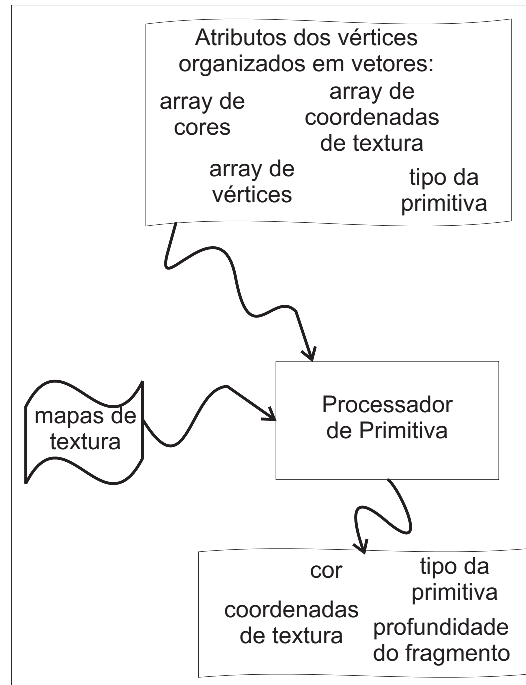

Figura 2.4: Entrada e saída de dados no processamento de primitivas

PUC-Rio - Certificação Digital Nº 0510966/CA

fazer outro acesso de textura, o que é chamado de ("dependent texture read") ou ainda usar uma textura para fazer operações de "Lookup Table".

2.2 Computação de propósito geral em GPU via Pipeline Gráfico

A utilização do hardware gráfico para computação de propósito geral requer consideráveis reformulações de algoritmos e estruturas de dados. Nesta seção abordaremos o uso da GPU para processamentos não gráficos, via programação de seu pipeline gráfico, ou seja, pelo uso de shaders, e na Seção 2.3 abordaremos seu uso com linguagens formuladas propositalmente para a computação genérica.

A área de pesquisa denominada Computação de Propósito Geral em GPU (do inglês, "General-Purpose Computation on GPUs"- GPGPU), representada pela comunidade de mesmo nome (GPGPU community), procura vencer diferentes desafios para a proposição de algoritmos tradicionalmente sequenciais e implementados em CPU para formulações paralelas coerentes com a arquitetura de processadores paralelos de fluxos (stream) de dados da GPU.

A utilização do pipeline gráfico oferece duas formas distintas de paralelismo, conhecidas como paralelismo de tarefas ("task parallelism") e de dados ("data parallelism"). O paralelismo de tarefas é caracterizado pela execução em paralelo de diferentes processos. O paralelismo de dados utiliza-se da organização de uma coleção de registros semelhantes em um fluxo de dados ("data

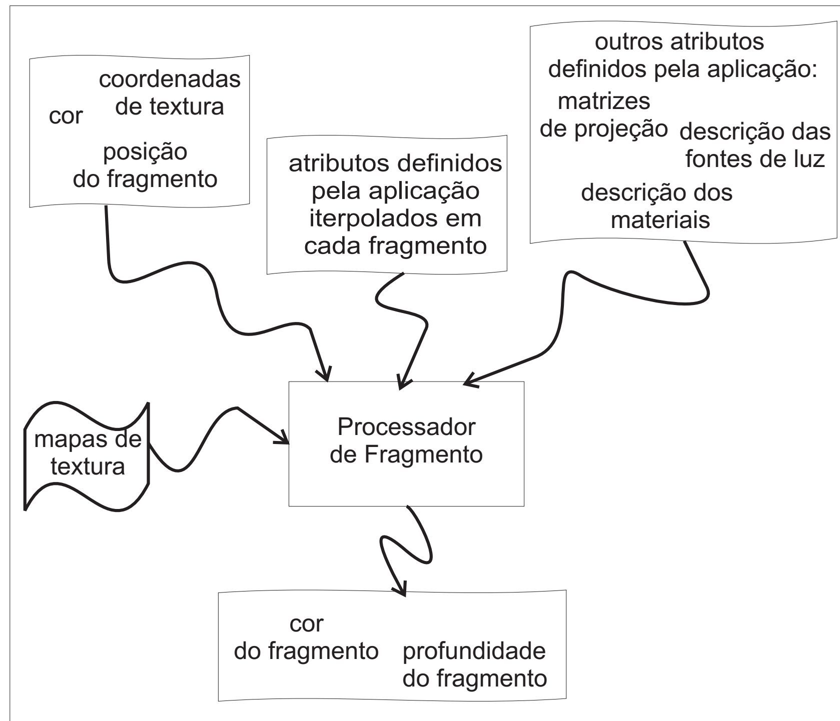

Figura 2.5: Entrada e saída de dados no processamento de fragmentos

stream") para executar computações similares em paralelo sobre múltiplos dados do fluxo. Normalmente, tais computações são independentes e com pouca comunicação entre si, mas executadas de forma a saturar muitas ALUs ("arithmetic logic unit").

Historicamente, as unidades de processamento das GPUs eram distribuídas entre os diferentes tipos de operação do pipeline gráfico, com determinada quantidade de unidades de processamento dedicada ao processamento de vértices e outra ao de fragmentos. Tipicamente, mais núcleos eram dedicados ao processamento de fragmentos do que ao de vértices, por ser comum uma grande diferença entre o número de fragmentos e o de vértices manipulados durante o processo de renderização.

Mais recentemente (a partir da série 8.0 da NVIDIA em 2007 (GeForce 8800 Technical Brief)), foram inseridas arquiteturas com múltiplos processadores independentes, com núcleos de processamento de ponto flutuante simples ("single-precision floating point"). Esse modelo apresenta uma arquitetura unificada, onde cada processador independente têm a capacidade de operar shaders de qualquer dos tipos, permitindo que os núcleos de processamento sejam dinamicamente alocados, de acordo com a demanda da aplicação.

Uma vez que as linguagens de shader não possuem códigos específicos à execução paralela, essas duas formas de paralelismo ocorrem de modo transparente ao programador. O paralelismo de tarefas ocorre pela distribuição

PUC-Rio - Certificação Digital Nº 0510966/CA

das etapas do pipeline em diferentes núcleos de processamento, enquanto o de dados ocorre pelo uso de mais de uma unidade de processamento para uma determinada etapa. Portanto, a escrita de shaders para computações genéricas beneficia-se de tais formas de paralelismo, uma vez que os shaders são computados em paralelo sobre múltiplos dados, também em paralelo.

O paralelismo de dados ("data parallelism"), permitindo que computações similares sejam aplicadas ao mesmo tempo a múltiplos elementos de um fluxo de entidades e a independência entre computações de diferentes entidades são as duas características das aplicações gráficas que mais influenciaram a arquitetura da GPU (GPU Gems 2). A medida utilizada para quantificar tais características em uma aplicação é a relação entre computações e a largura de banda ("bandwidth") e é chamada de intensidade aritmética. Mais formalmente, é obtida pela razão entre o número de operações e o de palavras transferidas.

Ao avaliar a adequação de aplicações para uma implementação via GPGPU consideram-se positivas as seguintes características: necessidade de processamentos similares sobre grandes volumes de dados; viabilidade de abordagens com alto grau de paralelismo; dependência mínima entre os elementos de processamento; alta intensidade aritmética; e grande quantidade de computação a ser processada sem a intervenção da CPU.

Apesar de existirem computações de diferentes propósitos, apresentamos uma formulação geral de um programa de GPGPU na tentativa de estruturar uma computação paralela sobre um fluxo de dados. A Figura 2.6 ilustra o processo de computação genérica via pipeline gráfico descrito.

PUC-Rio - Certificação Digital Nº 0510966/CA

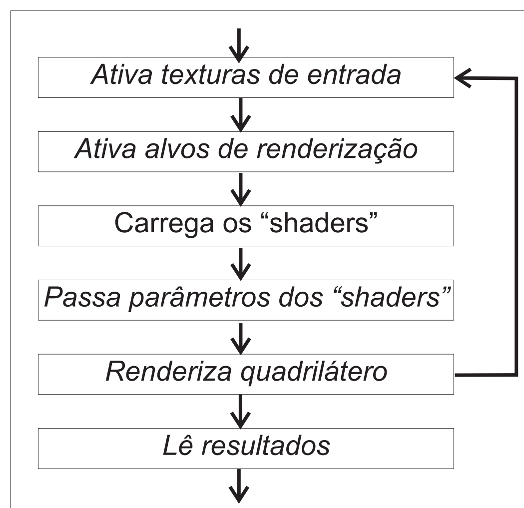

Figura 2.6: Etapas de uma renderização para computação genérica

Supondo que a computação é executada sobre um conjunto de dados de entrada, inicialmente deve ser feita uma transferência desses dados para a

memória da placa gráfica para que possam ser manipulados pelos shaders.

O tipo de memória mais usado para armazenar grandes volumes de dados de entrada das computações em GPU é a memória de texturas. As texturas podem ser pensadas como memórias somente de leitura (GPU Gems 2). Portanto, quando se deseja fornecer dados para a computação via uso de texturas, antes de iniciar a computação, os dados devem ser repassados da CPU para a GPU na forma de um array contendo os dados para a textura.

Enquanto em uma codificação sequencial os programas são escritos usando laços de repetição ("for", "while", "do while"), para executar computações similares sobre um conjunto de registros, na GPU a computação sobre um único registro é descrita em um shader (Figura 2.7). Para fazer com que a computação processe os diversos registros, ao invés de um laço de repetição, é preciso fazer com que o "shader"seja executado diversas vezes, isto é, que o shader seja aplicado a um fluxo de registros da entidade apropriada ao tipo do shader.

PUC-Rio - Certificação Digital Nº 0510966/CA

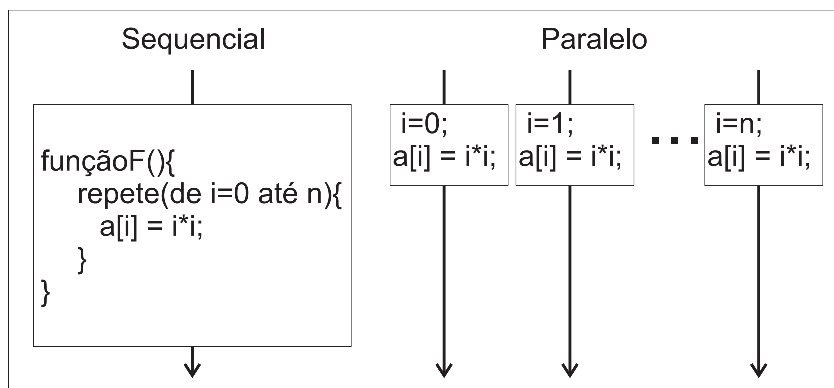

Figura 2.7: Laço sequencial × computações em fluxos de entidades

A forma mais comum de criar um fluxo no pipeline gráfico para processamento genérico é modelar uma cena contendo apenas um quadrilátero ("quad") com face paralela ao plano da tela, projetado ortograficamente, com dimensões determinadas pelo tamanho da saída desejada.

Ao renderizar tal quadrilátero, um fluxo de fragmentos é criado pela etapa de rasterização, na qual o quadrilátero será amostrado em cada pixel coberto da tela. A distribuição dos fragmentos gerados pela rasterização dessa primitiva deve criar uma associação de um para um com a distribuição dos dados que se deseja computar. Para isso, as coordenadas dos vértices especificadas via API gráfica devem estabelecer o intervalo da computação, ou seja, o tamanho do fluxo de saída (ver Figura 2.8). Como o rasterizador possui a capacidade de gerar muitos elementos a partir de poucos elementos (a primitiva que recebe), ele pode ser pensado como um amplificador de dados (GPU Gems 2).

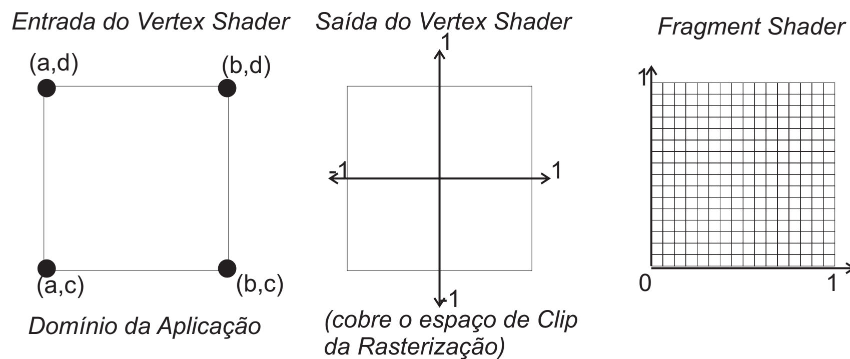

Figura 2.8: (da esquerda para a direita) Mapeamento do Domínio da Aplicação ao seu reposicionamento pelo Vertex Shader e a distribuição das saídas do Fragment Shader

Com o fluxo criado, a computação desejada sobre cada registro, representado como um fragmento, é feita usando o "fragment shader" ativo, que representa um único procedimento a ser aplicado a todos os elementos. As posições dos fragmentos indexam os dados gerados e são comumente utilizadas para acessar os dados de entrada armazenados em texturas. Para isso, a API gráfica pode ser usada para associar, aos quatro vértices do quadrilátero, coordenadas de textura, de forma a corresponder ao domínio da computação (ver Figura 2.8). Tais coordenadas também são interpoladas pelo rasterizador, associando a cada fragmento a coordenada de textura apropriada. Em diferentes aplicações pode ser interessante associar coordenadas de textura independentes para cada textura de entrada ou, ainda, computar tal coordenada no próprio "fragment shader", uma vez que o "fragment shader" é livre para fazer acesso aleatório de textura.

Cabe ressaltar que, nesse processo, a computação genérica formulada é executada por uma chamada ao pipeline de renderização e, portanto, o resultado obtido é armazenado na imagem gerada pelo processo de renderização. A saída desse processo é tradicionalmente associada à memória do "frame buffer", residente em memória da GPU e responsável pelo conteúdo visível da tela. Uma alternativa importante é associar a saída do processo de renderização a buffers especiais de textura na memória da placa. Essa técnica é denominada renderização para textura ("render-to-texture") e a escolha do armazenamento da saída gerada é feita pela alteração dos alvos de renderização ("render targets") via API gráfica. Uma vez que o resultado esteja armazenado em memória de textura ao fim da renderização e, portanto, já residente em memória da placa, pode vir a ser usado diretamente em processamentos subsequentes em GPU.

Quando necessário, o resultado pode ser passado à CPU usando instruções apropriadas da API gráfica utilizada (OpenGL ou DirectX3D), pela cópia da memória apropriada da GPU para a CPU. Em outros casos, o resul-

PUC-Rio - Certificação Digital Nº 0510966/CA

tado pode ser usado como entrada para outros processamentos de computação inicializados com novas chamadas a renderização e possíveis trocas de shaders, caracterizando uma abordagem com múltiplas passadas no pipeline.

Uma observação importante desse modelo é a concentração do processamento nos processadores de fragmento. Em arquiteturas com processadores dedicados às diferentes etapas, é comum que processadores de vértices fiquem desocupados, ainda que estejam em menor número do que os processadores de fragmento.

Revisando o modelo apresentado de forma mais abstrata, podemos identificar alguns conceitos comumente usados na área de processamento de fluxos. Como já foi exposto, um fluxo ("stream") é uma coleção ordenada de registros de um mesmo tipo a serem processados por computações similares.

A formulação de uma aplicação deve ser iniciada pelo programador, identificando as partes que possam ser computadas de maneira paralela. A aplicação é, então, repartida em seções paralelas independentes que podem ser computadas na forma de tarefas paralelas ou em múltiplas passadas. Cada uma dessas seções terá sua computação efetuada por um operador chamado "kernel", descrevendo o algoritmo na forma de operações a serem efetuadas a um único elemento do fluxo de cada vez.

Ao codificar um programa, são comuns manipulações de memórias globais nas formas de leitura e escrita. No modelo apresentado, a memória global também é considerada como um fluxo de dados e é utilizada para fornecer dados de entrada pré-carregados (ou produzidos) na memória da placa na forma de textura, ou como armazenamento dos resultados ao término do processo de renderização.

A operação de leitura indireta da memória global em uma posição arbitrária ($p = a[i]$) é chamada operação de "gather"(coleta) de memória pode ser feita naturalmente nos shaders pelo acesso de textura usando coordenadas de textura computadas, ou seja leitura de textura dependente (denominada "texture fetch") e pode ser usada para acessar estruturas de dados e outras informações armazenadas nessa memória. Atualmente as etapas programáveis possuem acesso a textura, mas historicamente, apenas o "fragment shader"possuia a capacidade de fazer gather. Esse acesso à memória global via amostragem de texturas pode ser feito inúmeras vezes dentro de um "kernel", inclusive a texturas distintas.

A operação de escrita indireta da memória global em uma posição arbitrária ($a[i] = p$) é chamada operação de "scatter"(dispersar) de memória. É uma operação fundamental para a construção de diversas estruturas de dados e tradicionalmente disponível em CPU. Nos shaders usuais o armazenamento

PUC-Rio - Certificação Digital Nº 0510966/CA

em memória global só é permitido na emissão dos valores de saída do shader, em endereços pré-estabelecidos. Para disponibilizar aos shaders a operação de "scatter"em memória global, seria necessário o suporte a operações de escrita dependente, ou seja, em endereços de memória calculados arbitrariamente durante o processamento.

Em um "fragment shader"o endereço de saída das informações de um fragmento não pode ser alterado e é pré-determinado na criação do fragmento pelo rasterizador, associado a posição do pixel correspondente ao fragmento. Portanto, não é capaz de efetuar "scatter"de memória.

O endereço de memória onde um "vertex shader"armazena as informações do vértice processado é pré-estabelecido no fluxo de saída. Entretanto, seu processamento pode alterar a informação descrevendo a posição 3D do vértice, da qual decorre a posição dos pixels cobertos pela primitiva correspondente ao vértice, amostradas pelo rasterizador. Portanto, considera-se que o vertex shader é capaz de fazer "scatter"de memória, uma vez que ao alterar a posição 3D do vértice são alteradas as posições dos fragmentos gerados.

Ao escrever um processamento que originalmente requisitaria uma operação de escrita indireta, são usados alguns artifícios para reescrever o problema, tais como, reformular o problema em termos de uma leitura (a operação passa a ser formulada com operações sobre a posição em que se deseja escrever), ou como usar marcações com o endereço onde se deseja escrever e em seguida usar um "vertex shader"para posicionar corretamente o dado, entre outros.

Observe que as conotações de coleta ("gather") e de dispersão ("scatter") estão associadas à posição do elemento em processamento. Portanto, a leitura está associada à idéia de trazer ou coletar informações de outras posições para o elemento corrente, enquanto que a escrita está associada a idéia de levar ou dispersar informações por outras posições que não a original.

2.3 "GPU Computing": Computação em GPU

O uso da GPU para computação genérica foi iniciado com a programação de shaders do pipeline gráfico, exigindo um conhecimento gráfico do programador, conforme detalhado na Seção 2.2. O termo GPGPU pode ser encontrado na literatura com a conotação de especificar a utilização da GPU para computação genérica, via API gráfica (Che07, Che08). Com a introdução de linguagens desenvolvidas especificamente para computação genérica em GPU, esta nova forma de programação do hardware gráfico, vem sendo diferenciada pela denominação Computação em GPU ("GPU Computing"). Neste contexto,

PUC-Rio - Certificação Digital Nº 0510966/CA

duas iniciativas se destacam: a API de programação genérica em GPU desenvolvida pela nVidia, denominada CUDA ("Compute Unified Device Architecture") (CUDA08) e a API desenvolvida pela AMD, chamada CTM ("Close to metal") (CTM08, Hen07). Nesta seção apresentamos uma visão geral sobre a API de programação genérica em GPU CUDA, escolhida pela sua maior aceitação.

A proposta de CUDA é oferecer um ambiente de programação para GPU que permita a execução de grandes quantidades de threads paralelas codificadas em uma linguagem similar à linguagem de programação C, a serem processadas em paralelo sobre os elementos de um domínio, eliminando a necessidade de mapear as aplicações através do pipeline gráfico e das APIs gráficas.

CUDA parte de uma abstração da CPU e GPU como dispositivos "host"e "device", definindo os respectivos papéis. A CPU ("host") é vista como um processador de controle, responsabilizado por configurar parâmetros, inicializar dados e executar as partes seriais do programa, ou que não sejam coerentes a uma formulação de paralelismo de dados. A GPU ("device") é vista como um co-processador, composta de conjuntos de multiprocessadores, cujo trabalho é acelerar partes que possam ser formuladas como computações paralelas de dados.

Assim como no modelo de programação genérica em GPU via pipeline gráfico, anteriormente apresentado, os procedimentos, que tradicionalmente eram descritos como um laço de repetição que percorre o domínio de processamento, na organização paralela fornecida por CUDA, podem ser repartidos em computações independentes sobre os elementos do domínio da aplicação. Em CUDA, a função a ser aplicada aos elementos e processada na GPU é denominada "kernel"e cada execução sobre um elemento é denominada thread. Para executar as tarefas de múltiplas threads concorrentemente, CUDA oferece uma estruturação para o domínio de processamento em uma grade ("grid"), subdividida em blocos ("blocks"), os quais, por sua vez, representam conjuntos de threads. Um grupo de um ou mais blocos é encaminhado a um processador que executa operações sobre múltiplos dados em paralelo e processa os blocos usando "time sharing".

Índices, tanto para indexar uma thread no interior de seu respectivo bloco, quanto para indexar o bloco corrente no interior da grade que define o domínio da aplicação, são fornecidos dinamicamente e associados a cada thread em execução. Esses índices podem vir a ser usados pelo programador na escrita de um "kernel"para computar mapeamentos relacionados à distribuição dos elementos do domínio, de acordo com uma lógica da aplicação.

PUC-Rio - Certificação Digital Nº 0510966/CA

CUDA apresenta um modelo de hierarquia de memória, cuja utilização é crucial para a performance das aplicações. Oferece acesso à memória DRAM da placa gráfica ("onboard memory") nos tipos constante, textura e global. O conteúdo dessas memórias é persistente entre chamadas de "kernels"de uma mesma aplicação e acessível ao "host", podendo por ele ser inicializada ou copiada ao término da aplicação de um "kernel".

Como memória de rápido acesso armazenada no chip de cada multiprocessador, a arquitetura CUDA oferece os seguintes tipos de memória: registradores e memória local associados às threads; uma área de memória compartilhada ("shared") acessível por bloco; caches somente de leitura, com cópias locais de trechos dos espaços da memória de textura e constante. A memória compartilhada permite que threads de um mesmo bloco compartilhem dados através dessa memória de alta velocidade.

O espaço de memória global permite que as operações em um "kernel"incluam leitura ("gather") e escrita ("scatter") indiretas de memória a um mesmo buffer localizado em memória global, aumentando a flexibilidade das aplicações e possibilitando a escrita de funcionalidades com menor uso de memória. Entretanto, acessos à memória da placa devem ser usados com precaução, por apresentarem significativas diferenças de tempo de transferência, quando comparadas à memória residente em chip.

Além dos ciclos de instrução utilizados para emitir uma instrução de acesso (leitura ou escrita) a uma memória qualquer, no caso de acesso à memória global ciclos extras são gastos até completar a instrução pela latência desse tipo de memória (CUDA08) (aproximadamente de 400 a 600 em uma nVidia GeForce 8800).

CUDA oferece mecanismos de sincronização por barreira para threads de um mesmo bloco que podem vir a ser utilizados juntamente com a memória compartilhada para uma comunicação entre tais threads. Threads de blocos distintos podem ser sincronizadas pelo término da chamada de função, ou seja, da execução do kernel em todos os elementos.

Threads de blocos distintos podem compartilhar dados via memória global, mas não há garantia da ordem de execução dos blocos de uma grade ("grid"), nem de que escritas na mesma posição de memória global por threads distintas concorrentes aconteçam. Por esse motivo, a memória global não é considerada um mecanismo de comunicação seguro. Operações atômicas, para o caso de GPUs que oferecem esse suporte, devem ser usadas com precaução, uma vez que são transformadas em operações seriais e a ordem de execução, ainda assim, não é garantida.

Em uma descrição geral sobre formulações de computação genérica se-

PUC-Rio - Certificação Digital Nº 0510966/CA

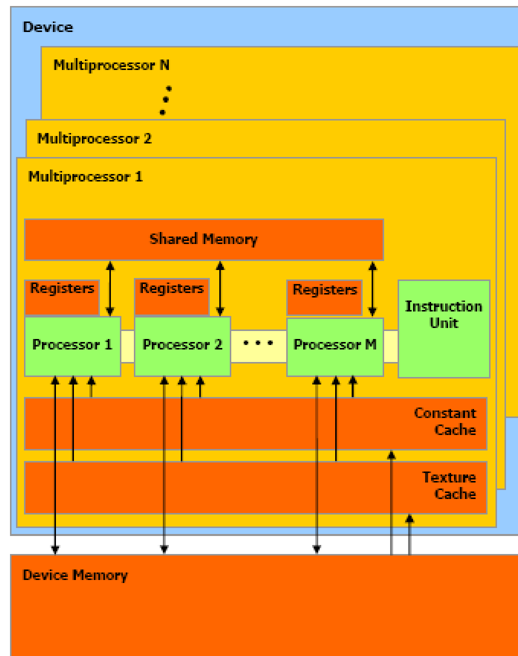

Figura 2.9: Hierarquia de Memória CUDA

guindo este modelo, inicialmente o programador define o domínio da computação na forma de uma grade de threads. Um kernel, composto de operações matemáticas e de acesso à memória compartilhada e global, é executado pelas threads. Cada thread é responsável por armazenar seu resultado em memória global. O buffer em memória global contendo os resultados pode, então, ser usado para futuras computações ou ser copiado para a memória do "host"(CPU).

Esse modelo geral de computação proposto por CUDA expõe a flexibilidade da GPU para a escrita dos "kernels", ao mesmo tempo que mantém as características que viabilizam a performance da GPU pelo modelo de execução de um programa sobre múltiplos dados (SPMD).

2.4 Arquitetura

Ao comparar os processadores das CPUs e GPUs modernos é interessante perceber que são formados basicamente dos mesmos componentes, os transistores. A tecnologia de fabricação de transistores avança no sentido de torná-los cada vez mais rápidos e menores que seus antecessores, ocupando áreas menores e operando de maneira mais eficiente, viabilizando a adição de mais transistores por chip.

Uma das projeções mais reconhecidas na área de tecnologia é chamada de Regra de Moore (Moo65) e é válida desde a criação dos circuitos integrados (1958). A Regra de Moore descreve uma projeção para a indústria de hardware,

PUC-Rio - Certificação Digital Nº 0510966/CA

afirmando que a quantidade de transistores que podem ser fabricados em um circuito integrado dobra a cada dois anos, aproximadamente.

São usados processos diferentes para fabricar, a partir dos transistores, componentes para processamento (aritmético e lógico) ou armazenamento (memória). A memória de acesso aleatório dinâmico (DRAM) tem sua capacidade avaliada pela largura de banda (do termo inglês "bandwidth"), compreendida pela capacidade de transferência de dado por unidade de tempo e pela latência, nesse contexto representado pelo espaço de tempo entre o pedido e o retorno de um dado da memória. De forma geral, enquanto a capacidade de processamento lógico e aritmético evolui rapidamente, a largura de banda, e principalmente, a latência de memória, não seguem tal ritmo (GPU Gems 2).

Em um nível mais alto, como alternativa à melhoria de performance pela diminuição do tempo de processamento de uma única "thread", o desenvolvimento moderno de microprocessadores concentra seus esforços em arquiteturas paralelas, pelo acréscimo de núcleos de processamento.

A forma com que as GPUs e CPUs são projetadas lida com esses desafios aplicados a interesses distintos. Os últimos anos demonstraram que não só a GPU está mais rápida do que a CPU, mas, também, que o crescimento de sua performance é mais acelerado (Figura 2.1). Modelos como a NVIDIA GeForce 8800 GTX podem sustentar 330 GFlops (bilhões de operações em ponto flutuante) e largura de banda de memória de streaming de 80 GB/s, medidas substancialmente maiores que as CPU multicore disponíveis, como o processador Intel Core Duo, com 48 GFlops e 10 GB/s, respectivamente.

As principais indústrias de hardware estão divididas em suas estratégias de paralelismo. A Intel e a Sun focalizam seus investimentos em soluções de "CPU many-core", consideradas soluções homogêneas, enquanto a IBM, a AMD e a nVidia concentram seus esforços em computação de fluxos ("stream computing"), soluções consideradas mais heterogêneas por envolverem CPUs e GPUs.

Uma vez que o processo de fabricação de semicondutores para os componentes da CPU e GPU é o mesmo, a disparidade está nas diferenças fundamentais de suas arquiteturas, conforme exposto nesta seção.

2.4.1
CPU

O modelo vigente de programação em CPU é conhecido como programação de fluxo de instruções ("Instruction Stream Programming") baseado na arquitetura de Von-Neumann, onde as instruções e os dados são armazenados na mesma memória. Durante o processamento, os dados necessários para

PUC-Rio - Certificação Digital Nº 0510966/CA

a execução de uma instrução são carregados para um cache de memória, se já não tiverem sido carregados anteriormente. A arquitetura de von-Neumann é extremamente flexível, mas o fato de ter a sequência de dados determinada pela sequência de instruções leva ao tratamento ineficiente de operações uniformes em grandes blocos de dados.

O núcleo de processamento da CPU é otimizado para alta performance em código sequencial, com vários transistores dedicados a extrair paralelismo no nível de instrução, através de técnicas como predição de desvios condicionais ("branch prediction") e execução fora de ordem ("out-of-order execution"). De forma geral, os projetos de arquiteturas de CPU atendem a demandas de execução de tarefas de códigos complexos e sequenciais sobre pequenas quantidades de dados, nos quais se espera baixa latência (amortizada por tratamentos com caches de instrução e dados, entre outras técnicas).

O paralelismo oferecido pelas CPUs multicores (vários núcleos) foi desenvolvido com o objetivo de atender às mais diferentes aplicações em paralelo. Um processador multicore combina dois ou mais núcleos ("cores") independentes em um único bloco de um único circuito integrado (IC -"integrated circuit"). Nesse modelo de microprocessador, técnicas de multiprocessamento são implementadas em um único dispositivo físico. Tal dispositivo pode ou não compartilhar caches de memória entre seus núcleos. Compartilham as conexões para o resto do sistema, mas o fundamental é que cada núcleo ("core") implementa independentemente otimizações e controles como execução superescalar, pipelining, e multithreading. Essa independência permite melhorar o tempo obtido em processamentos de múltiplas tarefas por sua capacidade de efetivamente executar dois programas ao mesmo tempo, um em cada núcleo. Portanto, permitem uma real paralelização de tarefas, em contraste com soluções de "multithreading"sobre apenas um núcleo de processamento, nas quais as tarefas devem compartilhar os recursos de tal núcleo.

2.4.2 GPU

A arquitetura do hardware gráfico teve seu desenvolvimento norteado pela natureza do processamento das aplicações gráficas 3D, induzindo a uma arquitetura especializada em computações massivas paralelas a dados.

Seguindo a demanda do pipeline gráfica, a arquitetura da GPU divide seus recursos de processamento entre as diferentes etapas do pipeline, oferecendo paralelismo de tarefas, na medida em que as etapas são tratadas em paralelo pelas diferentes unidades. A GPU é organizada de maneira que uma unidade de processamento responsável por uma etapa do pipeline repasse seus

PUC-Rio - Certificação Digital Nº 0510966/CA

resultados para alimentar outra unidade, designada à computação da etapa seguinte.

Esta organização em camadas de processamento permitiu que fossem desenvolvidos inicialmente hardwares de funcionalidade fixa, mas extremamente especializados. Assim, as unidades responsáveis por cada etapa ficaram livres para melhorar individualmente sua performance, explorando o paralelismo de dados existente internamente à correspondente etapa do pipeline. Com o aumento da flexibilidade nas etapas programáveis, o hardware fixo foi substituído por componentes programáveis. As demais etapas foram mantidas fixas, por serem mais eficientemente computáveis com hardware especializado (como por exemplo o hardware especializado em rasterização). Mesmo com essas mudanças, o hardware continua distribuindo os recursos de processamento entre tarefas e de forma encadeada para prover paralelismo de tarefas.

Essa organização em unidades responsáveis por executar operações semelhantes a um conjunto de elementos representando a mesma entidade gráfica permitiu que a GPU fosse composta de multiprocessadores SIMD ("Single instruction Multiple Data"), nos quais a cada ciclo de instrução vários elementos organizados em um fluxo são processados pelas mesmas instruções em paralelo.

Diferentemente da arquitetura de von-Neuman, na qual a sequência de dados é determinada pela sequência de instruções, a GPU foi projetada para processar sequências de dados determinados pelo fluxo de entidades gráficas e seus processadores são inicialmente configurados com as instruções que se deseja executar (sejam códigos de funcionalidades fixas, shaders ou "kernels"). Uma vez que o código é carregado, o fluxo de dados é processado com as mesmas instruções sobre múltiplos dados. Com menos transistores dedicados ao controle, a GPU não possui os mecanismos comumente usados na CPU para diminuir a latência e extrair paralelismo no nível de instrução, permitindo que a arquitetura das GPUs dediquem mais transistores para a computação e menos para controle. Por esse motivo, supondo uma mesma quantidade de transistores distribuídos segundo as arquiteturas da CPU e da GPU, a arquitetura da GPU atinge maior intensidade aritmética.

As características de tempo de latência e banda de memória da arquitetura das GPUs afetam diretamente a performance de algoritmos implementados em GPU. Otimizações podem ser feitas avaliando o impacto do custo de comunicação frente a possibilidade de computar valores, ao invés de buscá-los da memória. Outra forma de otimização é reformular as implementações para tolerar a latência, pelo preenchimento do tempo de espera de um pedido de dado na memória pela execução de processamentos aritméticos.

PUC-Rio - Certificação Digital Nº 0510966/CA

2.4.3 Desvio Condicional

Até agora foi discutido que os operadores de processamento na GPU ("shaders"ou "kernels"), executam processamentos independentes e similares nos elementos de um fluxo de dados. Atualmente, tanto as linguagens de "shader"quanto as de computação genérica permitem que seus códigos sejam escritos contendo desvios condicionais, aninhamentos e laços de repetição que podem fazer com que operações diferentes sejam executadas de acordo com análises dinâmica do processamento. Por esse motivo, é possível que o processamento de diferentes elementos do fluxo tomem diferentes caminhos dentro de um mesmo programa, obedecendo a um modelo de processamento de dados chamado SPMD ("single program multiple data").

O suporte à execução de diferentes caminhos para cada elemento requer substancial controle em hardware, na contramão do foco de projeto das GPUs, dedicado a computações aritméticas. Existem três abordagens disponíveis como mecanismos em hardware para controle de fluxo em GPU (Owe07b): predicação ("predication"), desvio condicional MIMD ("multiple instructions multiple data") e desvio condicional SIMD ("single instruction multiple data").

As arquiteturas com suporte do tipo chamado predicação não possuem verdadeiramente instruções de desvio condicional dependentes de dados. Seu tratamento em hardware é o de avaliar ambos os lados de um desvio condicional e então descartar os valores gerados pelo falso caminho, baseado na condição booleana de controle do desvio. Essa forma de tratamento é, portanto, extremamente cara.

A sigla MIMD representa a capacidade do hardware de oferecer suporte a desvios condicionais, de maneira que diferentes processadores sigam diferentes caminhos do programa. Esse tipo de tratamento foi introduzido pelos processadores de vértices das séries GeForce 6 e 7 e séries Quadro NV40 e G70 da nVidia.

Já no tratamento SIMD, todos os processadores ativos devem executar as mesmas instruções em um ciclo de instrução. Para viabilizar o tratamento de desvios condicionais, as GPUs agrupam os elementos a serem processados em paralelo em grupos de processamento. O tamanho desses grupos é denominado granularidade de ramificação ("branch granularity") e tem se tornado cada vez menor com o avanço tecnológico. Quando elementos de um mesmo grupo seguem um único caminho, apenas esse é processado. Caso contrário, o hardware computa ambos os lados do desvio condicional para todos os elementos de um grupo e faz predicação dos resultados, ignorando os caminhos falsos. Portanto, impõe penalidades na performance apenas nos casos de desvios condicionais

PUC-Rio - Certificação Digital Nº 0510966/CA

incoerentes entre elementos de um mesmo bloco. Portanto, identificar desvios condicionais em que o processamento do fluxo segue caminhos distintos permite identificar possíveis impactos na eficiência de códigos que utilizam tais formulações.

2.4.4 Arquitetura Unificada

O modelo de shader unificado (USM 4.0 –"Unified Shader Model") foi introduzido pelo DirectX 10 com o intuito de promover uma melhor utilização dos recursos do hardware gráfico pela distribuição dinâmica das unidades de processamento. Este modelo substituiu as unidades de processamento especializadas em cada tipo de shader por um projeto unificado, permitindo que as unidades de processamento sejam capazes de operar sobre qualquer dos tipos de shader. Para isso, todas as unidades programáveis compartilham de um conjunto básico de instruções aritméticas e de acesso à textura e buffers de dados (Figura 2.10). Cabe ressaltar que cada shader programável continua a desempenhar funcionalidades específicas de sua etapa.

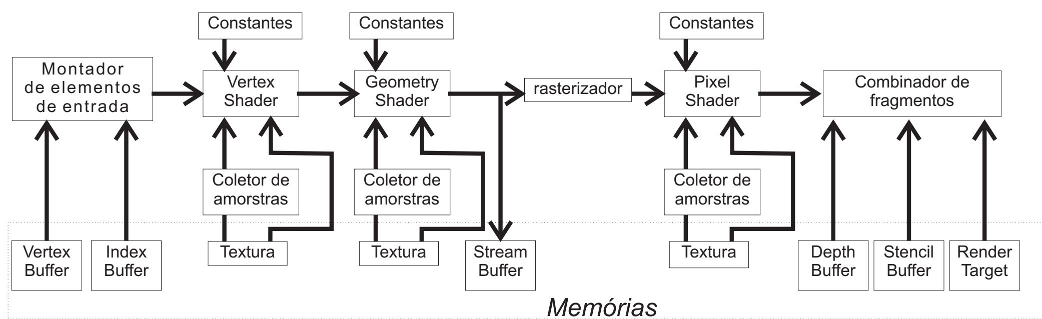

Figura 2.10: Arquitetura Unificada

Diferentes aplicações podem apresentar necessidades arbitrárias de processamento entre os shaders, concentrando o seu gargalo de processamento em uma única etapa do pipeline ou balanceando as operações entre elas. Com o projeto de arquitetura unificada, as aplicações que utilizam o pipeline gráfico se beneficiam de uma distribuição das unidades de processamento, transparente ao programador. Essa distribuição dinâmica evita o desperdício de unidades ociosas, comum nas arquiteturas dedicadas, ao custo de hardware mais complexo. Para as aplicações não-gráficas o benefício é ainda maior, pois permite utilizar todo o poder de processamento da GPU diretamente, sem ser necessária a divisão das operações da aplicação entre as unidades de hardware especializadas.

Entre as novas características importantes que o USM 4.0 impõe ao hardware destacam-se ainda: a capacidade de executar programas de até

PUC-Rio - Certificação Digital Nº 0510966/CA

65K instruções estáticas e ilimitadas instruções dinâmicas; um conjunto de instruções com suporte aos tipos inteiro e ponto flutuante 32 bits; suporte em hardware a leitura e escrita de memória global de forma direta ou indireta; e, suporte a controle de fluxo dinâmico (para laços ou desvios condicionais).

2.4.5 Visão geral: CPU x GPU

Em uma visão geral, o paralelismo das CPUs multi-cores permite a execução em paralelo de múltiplas threads que podem executar computações diferentes umas das outras. Todas as threads têm acesso total à memória e são livres para executar códigos distintos arbitrários. Para que esse modelo extremamente flexível seja eficiente, são exigidas técnicas sofisticadas de predição de código e de acesso à memória, que além de não escalarem com facilidade, demandam hardware dedicados a instruções de controle, ou seja, transistores que não estão sendo usados para processamento matemático nem lógico.

Por outro lado, o paralelismo das GPUs é baseado no processamento de fluxos e especializado no processamento de grandes demandas computacionais. Sua eficiência está relacionada à combinação de paralelismo de tarefas (execução em paralelo das tarefas do pipeline), e de dados (execução em paralelo de uma operação sobre múltiplos dados). Seus componentes de gerenciamento de memória da GPU são projetados para produzir um fluxo de dados ("streaming"), nos quais padrões de acesso lineares (2D) possam ser precarregados ("prefetched"), e assim esconder a latência de memória.

Sem os métodos de predição e controle oferecidos pela a CPU, a GPU é propícia a aplicações que tolerem latências de memória longas, em troca de produtividade. Fica assim evidenciado que enquanto as CPUs são eficientes em "time slicing"e "scheduling", por possuírem transistores dedicados a essas operações, as GPUs o são no processamento aritmético e lógico paralelo.

Em uma visão geral, pode-se classificar o paralelismo das CPUs multicore como um paralelismo "grosso"(nível de instruções), de threads pesadas ("heavyweight"), com complexidade arbitrária, nas quais a preocupação está em fornecer uma melhor performance por thread e aos mais diversos tipos de aplicação, mas com hardware dedicado a fornecer latências baixas. É ideal para tarefas dominadas por comunicação de memória e difíceis de paralelizar.

Já o paralelismo das GPUs é classificado como "fino"(nível de dados), com threads leves ("lightweight"). Quando analisada isoladamente, a performance por thread é pobre, mas compensada na produtividade do processamento sobre um conjunto de elementos, portanto, é amigável a aplicações de processamento de fluxos, aritmeticamente intensas.

PUC-Rio - Certificação Digital Nº 0510966/CA

Atualmente a computação em CPU oferece mais precisão que em GPU pelo suporte de tipos “double”, mas o aumento de precisão na GPU é esperado para um futuro próximo.

PUC-Rio - Certificação Digital Nº 0510966/CA