4
Programação Genérica na GPU

A GPU é cada vez mais usada para resolver problemas genéricos devido a seu alto poder de processamento. Porém, como este processador foi originalmente projetado para implementar o *pipeline gráfico*, os algoritmos mapeados para a GPU precisam se adaptar as operações deste *pipeline*. Neste capítulo, é descrito sucintamente as principais características de programação em placa gráfica e os principais mecanismos utilizados para mapear algoritmos genéricos para a GPU.

4.1.
Pipeline Gráfico

A computação na GPU segue um *pipeline* gráfico. Este *pipeline* foi desenvolvido para manter altas freqüências de computação através de execuções paralelas [15]. O *pipeline* gráfico convencional é composto por vários estágios parametrizáveis via API. No *pipeline* convencional, conforme ilustrado na Figura 6, a aplicação envia à GPU um conjunto de vértices. Estes vértices são transformados segundo matrizes de modelagem e visualização, depois são iluminados, projetados e mapeados para a tela. Após este conjunto de operações, a GPU combina os vértices para gerar algum tipo de primitiva (ponto, linha, triangulo, etc.). O rasterizador gera um fragmento para cada pixel que compõe a primitiva. Para cada fragmento, operações de mapeamento de textura, combinações de cores e testes de descarte podem ser feitos.

Com o surgimento das placas gráficas programáveis, alguns destes estágios podem ser substituídos por programas (*shaders*). Existem três tipos de programas que podem estar ativos em uma placa programável: o programa de vértices, o programa de geometria e o programa de fragmentos. O programa de geometria não será discutido neste trabalho, pois ainda é muito recente e não foi explorado na implementação deste trabalho.

Quando um programa de vértices ou de fragmentos é ativado, todos os estágios que ele substitui devem ser implementados (Figura 6). Não há como, por exemplo, implementar um programa de vértices para mudar apenas a iluminação.

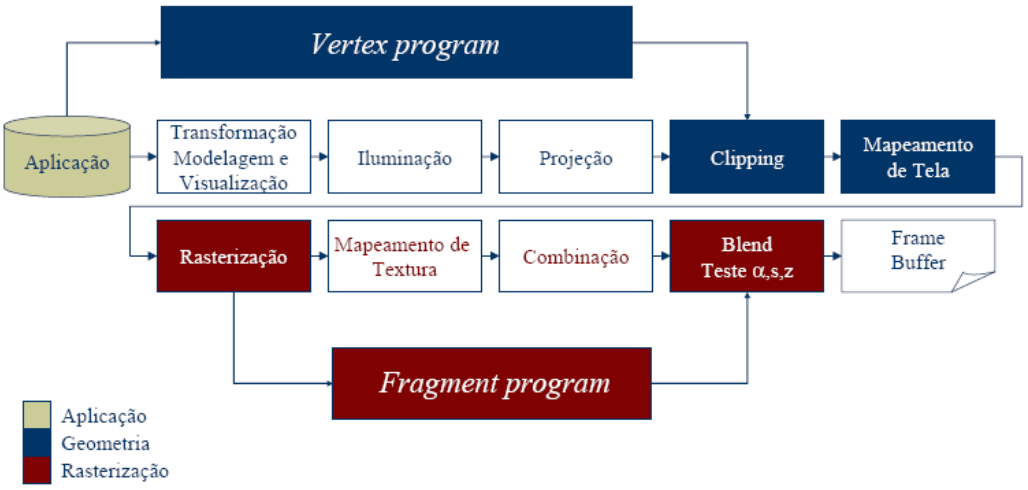

Figura 6 *Pipeline* Convencional x *Pipeline* Programável (sem considerar o programa de geometria)

4.2. Programação Genérica na GPU

O *pipeline* gráfico é dividido convencionalmente em estágios que fazem sentido em aplicações de computação gráfica. Porém, quando usamos a GPU para computação genérica, alguns daqueles estágios perdem o sentido. Por isso, o *pipeline* gráfico para programação genérica é dividido da seguinte maneira: operações de vértice, rasterização, operações de pixel e composição do resultado (Figura 7).

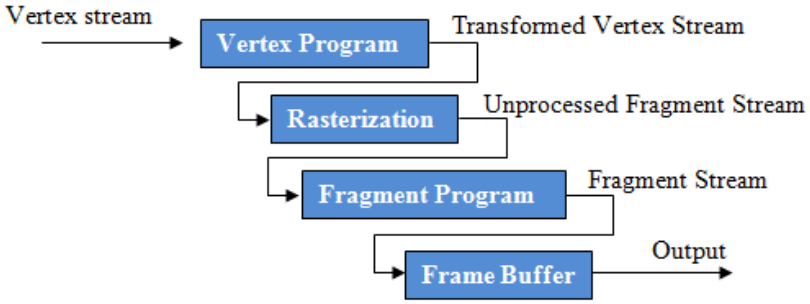

Figura 7 *Pipeline* simplificado para programação genérica em GPU.

4.2.1. Modelo de procesamento

A GPU é uma máquina paralela de processamento de *stream*. *Stream* é definido como uma seqüência de dados do mesmo tipo. Para ser eficiente, a GPU precisa fazer a computação de *streams* com grandes quantidades de elementos que sofram o mesmo tipo de operação.

Um programa de vértices ou de fragmentos opera sobre todos os elementos de um *stream*. Dentro desses programas, a computação de um elemento não depende dos outros elementos e a saída produzida é função apenas dos parâmetros de entrada do programa.

O modelo de processamento adotado no programa de fragmentos é chamado de SIMD (*single instruction multiple data*). Por isso instruções condicionais podem afetar bastante o desempenho de um algoritmo. Uma das técnicas para resolver o problema de instruções condicionais é criar vários *shaders*, uma para cada condição tratada.

4.2.2. Acesso à memória

O acesso à memória na GPU é feito de forma indireta. Os acessos randômicos de escrita e leitura em áreas da memória são chamados *scatter* e *gather*, respectivamente [15]. O suporte às operações de *scatter* representa a habilidade de escrever valores em uma posição qualquer da memória (x[i] = v), e o suporte às operações de *gather* representa a habilidade de ler valores de uma posição qualquer da memória (v = x[i]). Uma operação de *gather* na GPU é feita através de um acesso de textura. Como objetos de textura possuem um tamanho máximo, vetores com mais elementos do que este tamanho máximo tem que ser representados através de texturas bidimensionais ou tridimensionais. Portanto a tradução de endereços unidimensionais em bi ou tridimensionais é geralmente feita nos *shaders*.

A operação de *scatter* não pode ser feita no programa de fragmentos. Um programa de fragmentos só pode escrever na posição em que o fragmento será mapeado no *framebuffer*. No programa de vértices, a operação de *scatter* é feita através da transformação da posição do ponto, gerando um fragmento na posição

onde ser deseja escrever. Vale lembrar que apenas uma operação de *scatter* é feita por vértice.

4.2.3. Entrada e saída de dados

Um algoritmo é estruturado na GPU em passadas de *rendering*. Alguns algoritmos são feitos em uma passada e outros requerem várias passadas de *rendering*. Uma passada de *rendering* é um conjunto de operações que vão desde o envio de dados da CPU para a GPU até a saída produzida pela GPU. Os dados de entrada são passados através de atributos de vértices e/ou texturas. As texturas podem ser acessadas tanto no programa de vértices quanto no programa de fragmentos.

Os vértices passados como entrada são processados por um programa de vértices para compor uma primitiva (ponto, linha, triângulo, etc.). O rasterizador gera um fragmento para cada pixel que compõe a primitiva, interpolando valores de posição, cor e outros. Cada fragmento gerado é processado pelo programa de fragmentos. A saída do programa de fragmentos, que pode ser composta com o *framebuffer* corrente, gera o resultado de uma passada de *rendering*. Esta saída pode ser enviada para a tela ou ser armazenada numa textura de saída, que pode posteriormente servir de entrada para uma nova passada de *rendering*.

Para combinar ou priorizar os resultados produzidos pelo programa de fragmentos, testes de descarte (*z, alpha e stencil*) e operações de combinação (*blend*) podem ser usados. Podemos ainda usar testes de oclusão (*occlusion query*) para verificar a necessidade de passadas de *rendering* adicionais.

A GPU ainda possui limitações quanto ao número de texturas de entrada, número de texturas de saída, número de constantes, número de registradores utilizados e número de instruções dos *shaders*. Estes limites vêm aumentando com a evolução do processador gráfico.

Visando a programação genérica, novas funcionalidades estão sendo acopladas à GPU, como suporte a aritmética de inteiros, vetores de maior dimensão e a habilidade de gerar primitivas de dentro do *pipeline* gráfico (programa de geometria).