GPU – Graphics Processor Units

David Reis, Ivan Conti, Jeronimo Venetillo

Resumo

GPUs são processadores especializados em operações relacionadas com computação gráfica 3D. São extremamente poderosos devido a sua arquitetura paralela e sua eficiência, tanto no acesso a memória como nas operações vetoriais e de interpolação. Atualmente, devido o grande aumento de flexibilidade da arquitetura e das linguagens de programação, as GPUs estão sendo usadas para substituir a CPU na resolução de diversos algoritmos clássicos (GPU for Generic Programming - GPGPU). O uso cada vez mais freqüente da GPU para programação genérica é devido à grande eficiência que o processador gráfico possui em determinados tipos de operações, superando o desempenho da CPU.

1. Introdução

A GPU foi desenvolvida para ter um alto poder de processamento em operações relacionadas com computação gráfica 3D. Devido ao alto investimento no mercado de jogos, as GPUs se tornaram o hardware mais poderoso por unidade monetária. Com a inclusão das GPUs e o crescimento de seu desempenho, a CPU pode preocupar-se mais com simulações físicas, inteligência artificial e funcionamento lógico do jogo e deixar os algoritmos de renderização por conta da GPU.

Além dos algoritmos de renderização, a GPU tem sido usada também para programação genérica. Apesar do modelo computacional utilizado não ser adaptável a todo o tipo de problema, a GPU tem sido uma alternativa interessante em várias aplicações, principalmente nas que são *arithmetic-intensive*.

2. Histórico

Antes da criação das GPUs, as aplicações gráficas (em especial os jogos) eram baseados apenas em polígonos (Figura 1 - a), ou seja, na geometria da cena. Com criação da primeira placa gráfica comercial, a 3dxf voodoo, em 1995 houve a primeira grande revolução dos jogos. Esta placa gráfica era capaz de fazer mapeamento de textura em geometrias e possuía o algoritmo de z-buffer implementado em hardware. Agora cada geometria não precisava ser colorida individualmente (Figura 1 - b). Em 1998, com o lançamento da Nvidia TNT e da ATI Rage, houve duas mudanças: a implementação de multi-texturas (Figura 1 - c) e o uso do barramento AGP em vez do PCI. Estas mudanças tiveram um impacto de desempenho e não de funcionalidade. Era possível fazer multi-texturas usando mais de uma passada de rendering.

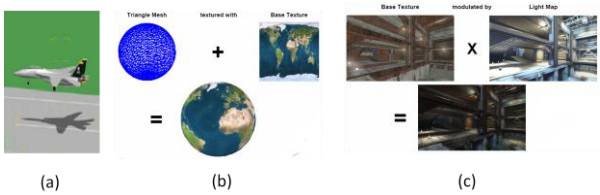
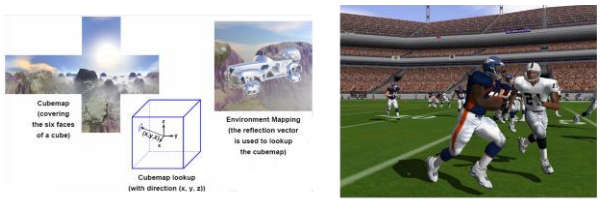


Figura 1. Primeira geração de gráficos

Em 1999 a Nvidia lança a GeForce 256 e a proclama como a primeira GPU do mundo (Figura 2 - a). Foi logo seguida pelas GeForce 2, Radeon 7500 e Savage 3D. A grande diferença destas placas para suas antecessoras foi a criação do conceito de *pipeline* gráfico. Neste conceito, uma GPU era responsável não só pela rasterização e pela texturização dos polígonos enviados, mas também era responsável pela transformação dos vértices dos polígonos e pela iluminação dos mesmos. Com isso, passou-se a enviar polígonos não mais nas coordenadas da tela, mas nas chamadas coordenadas de mundo.

Em 2001, surgem as placas gráficas programáveis (Figura 2 - b), com as GeForce 3, GeForce 4 e Radeon 8500. A programação só era possível no processador de vértices, e tinha diversas limitações quanto a sua funcionalidade. Além disso, foi criado o conceito de texturas 3D.



(a) (b)

Figura 2. Segunda geração de gráficos

Em 2002, o processador de fragmentos também se tornou programável (Figura 3 - a). Placas como GeForce FX, Radeon 9600-9800 e Radeon X600-800 faziam computação muito mais eficiente e flexível do que a geração anterior. Agora branches eram permitidos (porém limitados) e foram criados os *Multiple Rendering Targets*.

A série 6 da Nvidia lança o chamado Shader Model 3.0 em 2004 (Figura 3 - b). Esta série muda a programação em GPUs por permitir shaders mais longos, mais eficiência nos branches, suporte a precisão de 32 bits em ponto flutuante e acesso a textura no processador de vértices. A GPU teve também suporte a 64 bits de cores (16 bits por canal), o que permitiu que aplicações usassem composição de imagens com alta precisão e a criação de imagens com HDR (high dynamic range). Além disso, nesta época as placas gráficas começam a usar o barramento PCIe melhorando ainda mais o desempenho nas transferências entre CPU e GPU.

(a) (b)

Figura 3. Terceira geração de gráficos

Com a criação do barramento PCIe, em 2005 foram criadas as tecnologias SLI da Nvidia e Crossfire da ATI que permitiram interconectar múltiplas GPUs em um mesmo sistema. A distribuição de carga entre as GPUs pode ser feita manualmente ou automaticamente através do driver de vídeo.

Atualmente houve uma grande mudança por causa do lançamento das GPUs compatíveis com a nova especificação do DirectX 10 (Figura 4). A Nvidia GeForce 8 e a ATI Radeon HD2900 implementaram uma nova arquitetura. Agora existem três estágios de programação: o estágio de vértices, o novo estágio de geometria e o estágio de fragmentos. Outra mudança foi a unificação dos processadores de vértice e fragmentos em processadores únicos. Agora os recursos da GPU são alocados dinamicamente para o estágio que demanda mais recursos. Além disso foi adicionado suporte a tipos inteiros para ajudar na programação genérica em GPU.

DirectX 9 DirectX 10

Figura 4. DirectX 9 vs. DirectX 10

3. *Pipeline* Gráfico

A computação na GPU segue um *pipeline* gráfico. Este *pipeline* foi desenvolvido para manter altas freqüências de computação através de execuções paralelas. O *pipeline* gráfico convencional é composto por vários estágios parametrizáveis via API. No *pipeline* convencional (Figura 5), 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, os programas de vértices e de fragmentos podem substituir alguns destes estágios.

Quando um programa de vértices ou de fragmentos é ativado, todos os estágios que ele substitui devem ser implementados (Figura 5). Não há como, por exemplo, implementar um programa de vértices para mudar apenas a iluminação.

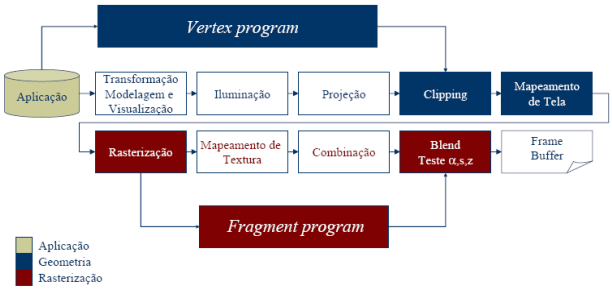


Figura 5. Pipeline Convencional x Pipeline Programável

Atualmente mais um estágio pode ser adicionado ao pipeline programável. É o estágio da geometria que não existe no pipeline fixo. Este estágio quebra o pipeline em duas partes e pode gerar novos vértices para realimentar o pipeline (Figura 6).

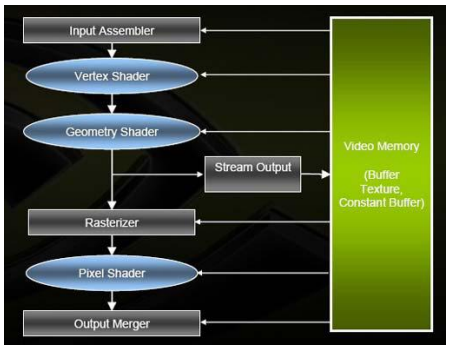


Figura 6. Novo pipeline

Estes programas que funcionam na GPU são chamados de *shaders*. Temos então o vertex shader, o geometry shader e o pixel shader.

4. Modelo de Processamento

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 shader 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.

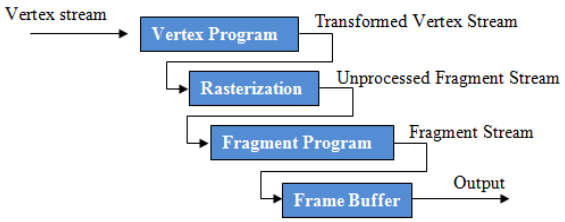


Figura 7. Modelo de processamento de streams

4.1. 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. 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. 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.

4.3. Arquitetura DirectX 10

As GPUs que implementam os requisitos do DirectX10 tem sua arquitetura um pouco diferente das GPUs anteriores.

Enquanto as GPUs anteriores possuíam processadores de vértices e processadores de fragmentos, as novas GPUs possuem processadores unificados. Isso quer dizer menos estágios internos de pipeline. O conceito de pipeline foi modificado para sair de um conceito de fluxo seqüencial para um modelo de fluxo em loop (Figura 8).

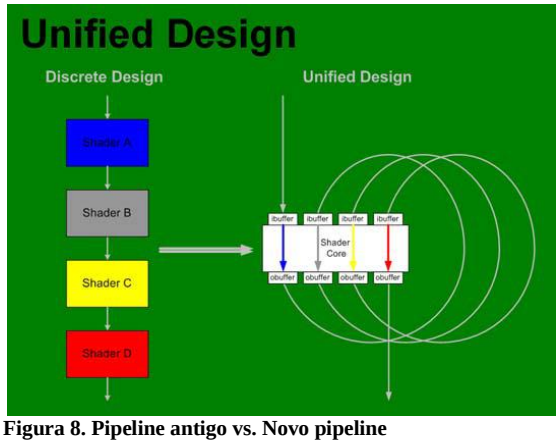


Figura 8. Pipeline antigo vs. Novo pipeline

As GPUs antigas usavam um modelo MIMD para o programa de vértices e SIMD para o programa de fragmentos, as novas GPUs usam conjuntos de processadores SIMD, e os conjuntos são alocados dinamicamente para o estágio que demande mais recursos computacionais.

As novas GPUs também oferecem suporte a inteiros, controle de fluxo totalmente dinâmico, um limite muito maior de instruções, constantes, registradores e tamanho de texturas.

O novo estágio de geometria difere dos outros estágios. Em vez de apenas processar um elemento, ele recebe um conjunto de dados e gera um outro conjunto de dados. Ou seja, enquanto os outros estágios fazem um processamento 1:1, este estágio faz um processamento N:M.

O estágio de geometria opera sobre um conjunto de vértices. Estes vértices possuem informações sobre vizinhança, definindo uma ou mais geometrias. A saída deste estágio também é um conjunto de vértices. Pode-se, por exemplo, enviar um conjunto de pontos independentes e, dentro destes estágios, criar conjuntos de linhas.

5. Característica do hardware

A GPU foi originalmente projetada para executar operações gráficas. Neste tipo de aplicação, geralmente a mesma operação é feita sobre vários dados diferentes. Por isso, a arquitetura da GPU é feita para operar em conjunto de streams de maneira paralela. Streams são definidos como grandes conjuntos ordenados de dados. Para fazer esta computação de maneira eficiente, o hardware foi implementado seguindo um pipeline gráfico. Este pipeline foi implementado em diversas unidades independentes que fazem uma computação paralela a nível de dados. Por isso, a GPU consegue manter altas freqüências de computação em execuções paralelas. Além dessa característica paralela, como o hardware é especializado, consegue obter uma eficiência muito maior que a de hardwares não especializados (como a CPU). Mais uma vantagem da GPU é que ela usa a maioria de seus transistores para a computação (datapath) e muito pouco para a parte de controle. Isso aumenta o poder de computação, porém torna o fluxo de um programa mais limitado. Outro aspecto da GPU é a velocidade de acesso a memória. Diferente da CPU, que prioriza a minimização da latência no acesso, a GPU tenta maximizar o throughput. Isso acontece porque a computação de GPU de stream se beneficia muito mais com o principio da localidade do que a computação genérica. Logo, a demora em se acessar um elemento não é tão importante como transferir um conjunto de elementos de maneira eficiente.

Com esse conjunto de características a GPU se tornou a potencia computacional de menor custo. Enquanto a CPU cresce numa taxa de 1.4 vezes por ano, a GPU cresce numa média de 1.7 a 2.3 vezes por ano. Comparando um Intel Pentium Extreme Edition 3.7 Ghz com um Nvidia GeForce 8800 Ultra, temos a GPU alcançando 576 GFLOPS contra 25.6 GFLOPS da CPU e uma taxa de transferência de memória de 86.4 GB/seg da GPU contra 8.5 GB/sec das CPU. Paralelo a isso, as funcionalidades oferecidas pelas GPUs vem crescendo a cada nova geração. Mudaram de pipelines fixos e com poucos formatos de dados a placas programáveis com uma grande variedade de formato de dados.

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, assim como a tecnologia utilizada na

construção da GPU, vêm aumentando com a evolução do processador gráfico (Figura 9).

	8800 Ultra	7900 GTX	6800 Ultra	5950 Ultra	4800 Ti
Tecnologia (nm)	90	90	130	130	150
Transistores (Milhoes)	690	278	222	130	63
Core Clock	612	650	400	475	300
Memory Clock	2x1080	2x800	2x550	2x475	2x325
Processadores de Vertice	128	8	6	3	2
Processadores de Fragmentos		24	16	4	4
ROPs Processors	24	16	16	8	8
Interface Memoria (bits)	384	256	256	256	128
Memory Bandwith (GB/sec)	103	51.2	35.2	30.4	10.4
Vertices/sec (Milhoes)		1400	600	356	136
Pixel Fill Rate (Bilhoes)		10.4	6.4	1.9	1.2
Texture Fill Rate (Bilhoes)	39.1	15.6	6.4	3.8	2.4
RAMDAC (MHz)		400	400	400	350
Power Consumption (W)	301	237	207	170	
GFLOPS	576	211	55	20	

Figura 9. Dados comparativos entre GPUs

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).

6. Conclusões

A GPU ainda é um componente muito recente na computação e permanece em constante e rápida evolução. O seu grande potencial pode ser identificado na grande quantidade de modificações que a sua arquitetura sofre a cada nova versão, seja na modelagem e dinâmica dos seus sub-componentes ou na tecnologia utilizada em sua construção. Além disso, a grande revolução no conceito e uso da GPU foi ainda mais recente: a programação genérica para o processador gráfico.

Desde o surgimento da idéia, os fabricantes de processadores gráficos vêm investindo na modificação da arquitetura da GPU, de forma a facilitar e ampliar o seu uso para programação genérica. De fato, esta aplicação do hardware gráfico é cada vez mais freqüente e próxima da realidade dos programadores convencionais, sendo utilizada como um meio de melhorar a eficiência no processamento de programas originalmente não-gráficos.

Atualmente, a GPU é o hardware mais barato para computação de alto desempenho (quando a tarefa a ser executada se aproveita do paralelismo explícito da GPU). O seu desempenho é bastante superior à CPU em diversos aspectos, o que pode ser verificado em simples comparações de taxa de GFLOPS (Figura 10) e performance de uma ordenação (Figura 11), por exemplo. Mas apesar do hardware gráfico estar ficando cada vez mais programável, ainda é necessária uma série de truques para garantir que um programa pode obter a melhor performance possível em uma GPU, que exigem o uso adequado de z-culling, texturas, branches, tipos de variáveis, etc. Além disso, não é qualquer aplicação que pode ser portada para a GPU, uma vez que esta operação exige que o problema possa ser remodelado como um problema gráfico.

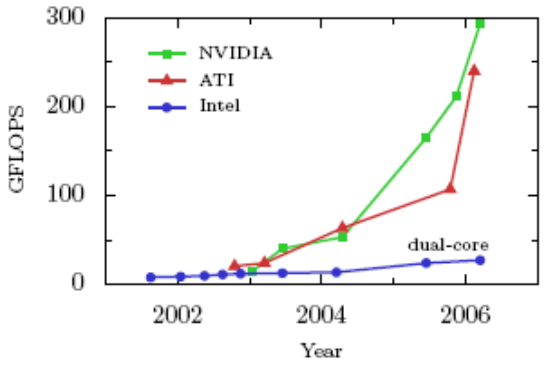


Figura 10. Performance das GPUs em comparação com a CPU

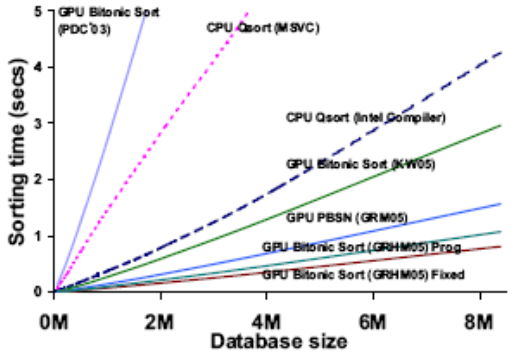


Figura 11. Desempenho comparativo dos processadores em uma operação "sort"

Com efeito, a programação genérica na GPU está amadurecendo rapidamente e, mais do que isso, os avanços no desenvolvimento do hardware da GPU e na sua programabilidade são um indicativo da tendência do uso mais generalizado de processadores com paralelismo nativo, representando uma área promissora nos campos acadêmico e industrial para os próximos anos.

7. Referências

[1] Godfrey Cheng. ATI Crossfire Press Update. Computex, 2006.

[2] NVIDIA Corporation. Geforce256 – the world's first gpu. 1999.
[3] NVIDIA Corporation. Nvidia cinefx architecture. 2002.
[4] NVIDIA Corporation. Nvidia cinefx architecture – real-time cinematic rendering. 2002.
[5] NVIDIA Corporation. Nvidia geforce 8800 architecture technical brief. 2006.
[6] Randy Fernando. Gpgpu: General-purpose computation on gpus. 2004.
[7] Randy Fernando and Cyril Zeller. Programming graphics hardware. 2004.
[8] ATI Technologies Inc. Asymmetric physics processing with ati crossfire. 2006.
[9] Emmett Kilgariff and Randima Fernando. The GeForce 6 Series GPU Architecture. 2005.
[10] David Kirk. Geforce3 architecture overview.
[11] John D. Owens; David Luebke; Naga Govindaraju; Mark Harris; Jens Kruger; Aaron E.
Lefohn; and Timothy J. Purcell. A survey of general-purpose computation on
graphics hardware. 2007.
[12] Bill Mark. Nvidia programmable graphics technology. 2002.
[13] John Owens. Streaming Architectures and Technology Trends. 2005.
[14] Derek Wilson. Ati radeon hd 2900 xt: Calling a spade a spade. 2007.