

A

A P É N D I C E

*Imaginação é mais
importante que
conhecimento.*

Albert Einstein,
On Science, 1930s

Gráficos e GPUs de computação

John Nickolls
Diretor de Arquitetura,
NVIDIA

David Kirk
Cientista Chefe,
NVIDIA

A.1	Introdução	569
A.2	Arquiteturas de sistemas GPU	572
A.3	Programando GPUs	576
A.4	Arquitetura de multiprocessador multithreaded	587
A.5	Sistema de memória paralela	597
A.6	Aritmética de ponto flutuante	601
A.7	Vida real: o NVIDIA GeForce 8800	605
A.8	Vida real: mapeando aplicações a GPUs	612
A.9	Falácia e armadilhas	626
A.10	Comentários finais	629
 A.11	Perspectiva histórica e leitura adicional	629

A.1 Introdução

Este apêndice focaliza a GPU — a onipresente **unidade de processamento gráfico** em cada PC, laptop, computador desktop e estação de trabalho. Em sua forma mais básica, a GPU gera gráficos 2D e 3D, imagens e vídeo que habilitam sistemas operacionais baseados em janelas, interfaces gráficas com o usuário, jogos de vídeo, aplicações de representação visual e vídeo. A GPU moderna que descrevemos aqui é um multiprocessador altamente paralelo, altamente multithreaded, otimizado para **computação visual**. Para oferecer interação visual em tempo real com objetos calculados via gráficos, imagens e vídeo, a GPU tem uma arquitetura gráfica e de computação unificada, que serve como um processador gráfico programável e uma plataforma de computação paralela escalável. PCs e consoles de jogo combinam uma GPU com uma CPU para formar **sistemas heterogêneos**.

unidade de processamento gráfico (GPU) Um processador otimizado para gráficos 2D e 3D, vídeo, computação visual e exibição.

computação visual Uma mistura de processamento gráfico e computação, que lhe permite interagir visualmente com os objetos computados através de gráficos, imagens e vídeo.

Breve histórico da evolução da GPU

Há quinze anos, não havia algo do tipo GPU. Os gráficos em um PC eram realizados por um controlador VGA (Video Graphics Array). Um controlador VGA era simplesmente um controlador de memória e gerador de vídeo conectado a alguma DRAM. Na década de 1990, a tecnologia de semicondutor avançou suficientemente para que mais funções pudessem ser acrescentadas ao controlador VGA. Em 1997, os controladores VGA estavam começando a incorporar algumas funções de aceleração tridimensional (3D), incluindo hardware para configuração e rasterização de triângulos (cortar triângulos em pixels individuais) e mapeamento e sombreamento de textura (aplicar “decalques” ou padrões aos pixels e misturar cores).

Em 2000, o processador gráfico de único chip incorporou quase todos os detalhes do pipeline gráfico da estação de trabalho de alto nível tradicional e, portanto, mereceu um novo nome além de controlador VGA. O termo GPU foi usado para indicar que o dispositivo gráfico tornou-se um processador.

Com o tempo, as GPUs tornaram-se mais programáveis, pois os processadores programáveis substituíram a lógica dedicada de função fixa enquanto mantinham a organização básica do pipeline gráfico 3D. Além disso, os cálculos se tornaram mais precisos com o tempo, progredindo da aritmética indexada até inteiros e ponto flutuante, até ponto flutuante de precisão simples, e recentemente para ponto flutuante de precisão dupla.

GPs tornaram-se processadores programáveis maciçamente paralelos, com centenas de cores e milhares de threads.

Recentemente, instruções de processador e hardware de memória foram acrescentadas para dar suporte a linguagens de programação de uso geral, e um ambiente de programação foi criado no sentido de permitir que as GPUs fossem programadas usando linguagens conhecidas, como C e C++. Essa inovação torna a GPU um processador totalmente de uso geral, programável e de muitos cores, apesar de ainda ter alguns benefícios e limitações especiais.

Tendências gráficas da GPU

interface de programação de aplicação (API) Um conjunto de definições de função e estrutura de dados que fornece uma interface para uma biblioteca de funções.

GPs e seus drivers associados implementam os modelos OpenGL e DirectX do processamento gráfico. OpenGL é um padrão aberto para a programação gráfica 3D, disponível para a maioria dos computadores. DirectX é uma série de interfaces de programação de multimídia da Microsoft, incluindo Direct3D para gráficos 3D. Como essas **interfaces de programação de aplicação (APIs)** possuem comportamento bem definido, é possível criar aceleração de hardware eficaz das funções de processamento gráfico definidas pelas APIs. Esse é um dos motivos (além de aumentar a densidade do dispositivo) para que novas GPUs estejam sendo desenvolvidas a cada 12 a 18 meses, o que dobra o desempenho da geração anterior nas aplicações existentes.

A duplicação frequente do desempenho da GPU possibilita novas aplicações, que não eram possíveis anteriormente. A interseção de processamento gráfico e computação paralela convida um novo paradigma para gráficos, conhecido como computação visual. Ela substitui grandes seções do modelo tradicional de pipeline gráfico de hardware sequencial por elementos programáveis para programas de geometria, vértice e pixel. A computação visual em uma GPU moderna combina o processamento gráfico e a computação paralela em novas maneiras, que permitem que novos algoritmos gráficos sejam implementados, uma porta aberta para aplicações de processamento paralelo inteiramente novas sobre GPUs predominantes de alto desempenho.

Sistema heterogêneo

Embora a GPU seja discutivelmente o processador mais paralelo e mais poderoso em um PC típico, certamente não é o único. A CPU, agora multicore, é um processador complementar, antes de tudo serial, para a GPU manycore maciçamente paralela. Juntos, esses dois tipos de processadores compreendem um sistema multiprocessador heterogêneo.

O melhor desempenho para muitas aplicações vem do uso da CPU e da GPU. Este apêndice o ajudará a entender como e quando dividir melhor o trabalho entre esses dois processadores cada vez mais paralelos.

GPU evolui para processador paralelo escalável

GPs evoluíram funcionalmente de controladores VGA fisicamente conectados, de capacidade limitada, para processadores paralelos programáveis. Essa evolução prosseguiu alterando o pipeline gráfico lógico (baseado em API) para incorporar elementos programáveis e também tornando os estágios de pipeline de hardware básicos menos especializados e mais programáveis. Por fim, fez sentido mesclar diferentes elementos de pipeline programáveis em um array unificado de muitos processadores programáveis.

Na geração de GPs GeForce 8-series, o processamento de geometria, vértice e pixel é executado no mesmo tipo de processador. Essa unificação permite uma escalabilidade incrível. Mais cores de processador programável aumentam a vazão total do sistema. Unificar os processadores também oferece balanceamento de carga bastante eficaz, pois qualquer função de processamento pode usar o array de processadores inteiro. Na outra ponta do espectro, um array de processadores agora pode ser montado com muito poucos processadores, pois todas as funções podem ser executadas nos mesmos processadores.

Por que CUDA e computação GPU?

Esse array de processadores uniforme e escalável convida a um novo modelo de programação para a GPU. A grande quantidade de poder de processamento de ponto flutuante no array de processadores GPU é muito atraente para solucionar problemas não gráficos. Dado o grande grau de paralelismo e a faixa de escalabilidade do array de processadores para aplicações gráficas, o modelo de programação para a computação mais geral deverá expressar diretamente o forte paralelismo, mas levando em conta uma execução escalável.

Computação GPU é o termo criado para usar a GPU para computação através de uma linguagem de programação paralela e API, sem usar a API gráfica tradicional e o modelo de pipeline de gráfico. Isso é contrário à técnica mais antiga da **General Purpose computation on GPU (GPGPU)**, que envolve programar a GPU usando uma API gráfica e pipeline gráfico para realizar tarefas não gráficas.

Compute Unified Device Architecture (CUDA) é um modelo de programação paralela escalável e plataforma de software para a GPU e outros processadores paralelos, que permite que o programador evite a API gráfica e interfaces gráficas da GPU e simplesmente programe em C ou C++. O modelo de programação CUDA tem um estilo de software SPMD (Single-Program Multiple Data – único programa, múltiplos dados), no qual um programador escreve um programa para uma thread que é instanciada e executada por muitas threads em paralelo nos múltiplos processadores da GPU. De fato, CUDA também oferece uma facilidade para programar também múltiplos cores de CPU, de modo que CUDA é um ambiente para escrever programas para todo o sistema de computador heterogêneo.

GPU unifica gráficos e computação

Com o acréscimo de CUDA e computação GPU às capacidades da GPU, agora é possível usar a GPU como um processador gráfico e um processador de computação ao mesmo tempo, e combinar esses usos nas aplicações de computação visual. A arquitetura do processador de suporte da GPU é exposta de duas maneiras: primeiro, implementando as APIs gráficas programáveis, e segundo, como um array de processadores altamente paralelos e programáveis em C/C++ com CUDA.

Embora os processadores de suporte da GPU sejam unificados, não é necessário que todos os programas de thread SPMD sejam iguais. A GPU pode executar programas de sombreamento gráfico para o aspecto gráfico da GPU, para processar geometria, vértices e pixels, e também executar programas de thread em CUDA.

A GPU é, na realidade, uma arquitetura de multiprocessador versátil, com suporte a uma série de tarefas de processamento. As GPUs são excelentes em gráficos e computação visual, pois foram projetadas especificamente para essas aplicações. As GPUs também são excelentes em muitas aplicações de vazão de uso geral, que são “primos de primeiro grau” dos gráficos, pois realizam muito trabalho paralelo, além de ter muita estrutura de problema regular. Em geral, elas são uma boa combinação para os problemas paralelos de dados (veja Capítulo 7), em especial os grandes, mas nem tanto para problemas menores e menos regulares.

Aplicações de computação visual da GPU

A computação visual inclui os tipos tradicionais de aplicações gráficas além de muitas aplicações novas. O escopo original de uma GPU era “tudo com pixels”, mas agora ele inclui muitos problemas sem pixels, mas com computação regular e/ou estrutura de dados. As GPUs são eficazes em gráficos 2D e 3D, pois essa é a finalidade para a qual elas foram projetadas. Deixar de oferecer esse desempenho da aplicação seria fatal. Gráficos 2D e 3D utilizam a GPU em seu “modo gráfico”, acessando o poder de processamento da GPU através das APIs gráficas, OpenGL™ e DirectX™. Jogos são construídos sobre a capacidade de processamento de gráficos 3D.

Computação GPU Usar uma GPU para computação através de uma linguagem de programação paralela e API.

GPGPU Usar uma GPU para computação de uso geral através de uma API gráfica tradicional e pipeline gráfico.

CUDA Um modelo de programação paralelo escalável e linguagem baseada em C/C++. Essa é uma plataforma de programação paralela para GPUs e CPUs multicore.

Além dos gráficos 2D e 3D, o processamento de imagens e vídeo são aplicações importantes para as GPUs. Estes podem ser implementados usando as APIs gráficas ou como programas computacionais, usando CUDA para programar a GPU no modo de computação. Usando CUDA, o processamento de imagens é simplesmente outro programa de array paralelo de dados. Na medida em que o acesso aos dados for regular e existir boa localidade, o programa será eficiente. Na prática, o processamento de imagem é uma aplicação muito boa para GPUs. O processamento de vídeo, especialmente a codificação e a decodificação (compactação e descompactação de acordo com alguns algoritmos padrão) é muito eficiente.

A maior oportunidade para aplicações de computação visual sobre GPUs é “romper o pipeline gráfico”. As primeiras GPUs implementavam apenas APIs gráficas específicas, embora com um desempenho muito alto. Isso era maravilhoso se a API aceitasse as operações que você queria fazer. Se não, a GPU não poderia acelerar sua tarefa, pois a funcionalidade inicial da GPU era imutável. Agora, com o advento da computação da GPU e CUDA, essas GPUs podem ser programadas para implementar um pipeline virtual diferente, simplesmente escrevendo-se um programa CUDA para descrever a computação e o fluxo de dados que se deseja. Assim, todas as aplicações agora são possíveis, o que estimulará novas técnicas de computação visual.

A.2

Arquiteturas de sistemas GPU

Nesta seção, analisamos as arquiteturas de sistemas GPU em uso comum hoje em dia. Discutimos as configurações do sistema, funções e serviços da GPU, interfaces de programação padrão e uma arquitetura interna básica da GPU.

Arquitetura heterogênea de sistema CPU-GPU

Uma arquitetura heterogênea de sistema de computação usando uma GPU e uma CPU pode ser descrita em um nível alto por duas características principais: primeiro, quantos subsistemas funcionais e/ou chips são usados e quais são suas tecnologias de interconexão e topologia, e segundo, quais subsistemas de memória estão disponíveis a esses subsistemas funcionais. Veja no Capítulo 6 uma base sobre os sistemas de E/S e chip sets do PC.

O PC histórico (por volta de 1990)

A [Figura A.2.1](#) é um diagrama em blocos de alto nível de um PC legado, por volta de 1990. A bridge norte (veja Capítulo 6) contém interfaces com alta largura de banda, conectando a CPU, memória e barramento PCI. A bridge sul contém interfaces e dispositivos legados: barramento ISA (áudio, LAN), controladora de interrupção; controladora de DMA; hora/contador. Nesse sistema, a exibição foi controlada por um subsistema de framebuffer simples, conhecido como VGA (Video Graphics Array), que foi conectado ao barramento PCI. Os subsistemas gráficos com elementos de processamento embutidos (GPUs) não existiam no panorama do PC de 1990.

A [Figura A.2.2](#) ilustra duas configurações comuns em uso atualmente. Estas são caracterizadas por uma GPU (GPU discreta) e CPU separadas, com respectivos subsistemas de memória. Na [Figura A.2.2a](#), com uma CPU da Intel, vemos a GPU conectada por meio de um link PCI-Express 2.0 de 16 pistas, para oferecer uma taxa de transferência máxima de 16GB/s (máximo de 8GB/s em cada direção). De modo semelhante, na [Figura A.2.2b](#), com uma CPU da AMD, a GPU está conectada ao chip set, também por meio de PCI-Express, com a mesma largura de banda disponível. Nos dois casos, as GPUs e CPUs podem acessar a memória um do outro, embora com menos largura de banda disponível que seu acesso às memórias conectadas mais diretamente. No caso do sistema da AMD, a bridge norte ou a controladora de memória é integrada ao mesmo die que a CPU.

PCI-Express (PCIe) Uma interconexão de E/S padrão do sistema, que usa links ponto a ponto. Os links têm um número configurável de pistas e largura de banda.

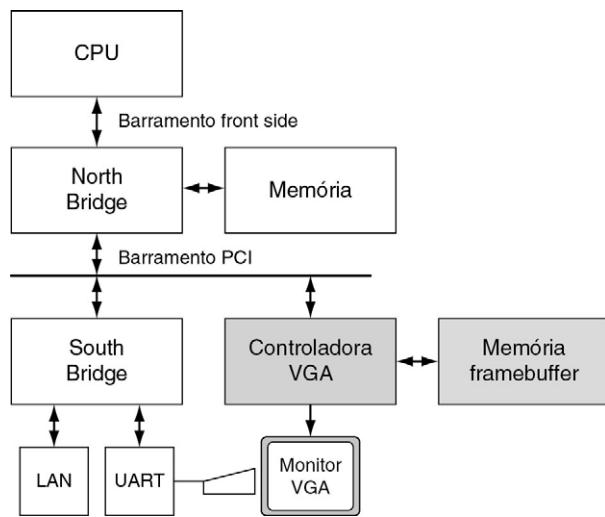


FIGURA A.2.1 PC histórico. A controladora VGA controla a exibição gráfica da memória do framebuffer.

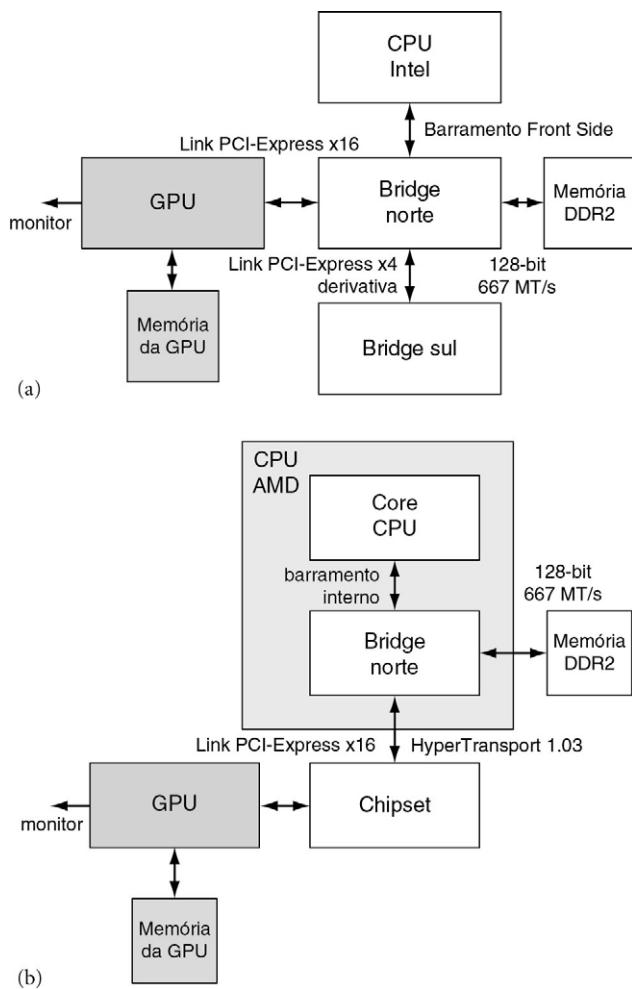


FIGURA A.2.2 PCs contemporâneos com CPUs Intel e AMD. Veja no Capítulo 6 uma explicação dos componentes e interconexões nesta figura.

Unified Memory Architecture (UMA) Uma arquitetura de sistema em que a CPU e a GPU compartilham uma memória de sistema comum.

Uma variação de baixo custo desses sistemas, um sistema **Unified Memory Architecture (UMA)**, usa apenas memória do sistema da CPU, omitindo a memória da GPU do sistema. Esses sistemas têm GPUs com desempenho relativamente baixo, pois seu desempenho alcançado é limitado pela largura de banda disponível da memória do sistema e latência aumentada do acesso à memória, enquanto a memória da GPU dedicada oferece alta largura de banda e baixa latência.

Uma variação do sistema de alto desempenho utiliza múltiplas GPUs conectadas, normalmente duas a quatro trabalhando em paralelo, com suas telas encadeadas em forma de margarida. Um exemplo é o sistema multi-GPU NVIDIA SLI (Scalable Link Interconnect), projetado para jogos de alto desempenho e estações de trabalho.

A próxima categoria de sistema integra a GPU com a bridge norte (Intel) ou chipset (AMD) com e sem memória gráfica dedicada.

O Capítulo 5 explica como os caches mantêm coerência em um espaço de endereço compartilhado. Com CPUs e GPUs, existem múltiplos espaços de endereço. As GPUs podem acessar sua própria memória local física e a memória física do sistema da CPU usando endereços virtuais que são traduzidos por uma MMU na GPU. O kernel do sistema operacional gerencia as tabelas de página da GPU. Uma página física do sistema pode ser acessada usando transações PCI-Express coerentes ou não coerentes, determinadas por um atributo na tabela de página da GPU. A CPU pode acessar a memória local da GPU através de um intervalo de endereços (também chamado de abertura) no espaço de endereços PCI-Express.

Consoles de jogos

Sistemas de console, como o Sony PlayStation 3 e o Microsoft Xbox 360 são semelhantes às arquiteturas de sistemas do PC, descritas anteriormente. Os sistemas de console são projetados para serem entregues com desempenho e funcionalidade idênticas por um espaço de tempo que pode durar cinco anos ou mais. Durante esse tempo, um sistema pode ser reimplementado muitas vezes para explorar processos de manufatura com silício mais avançados, e assim oferecer capacidade constante a custos ainda mais baixos. Os sistemas de console não precisam ter seus subsistemas expandidos e atualizados da forma como os sistemas de PC fazem, de modo que os principais barramentos internos do sistema tendem a ser customizados, em vez de padronizados.

Interfaces e drivers de GPU

Em um PC de hoje, as GPUs são conectadas a uma CPU por meio da PCI-Express. As gerações anteriores usavam **AGP**. As aplicações gráficas chamam funções de API OpenGL [Segal e Akeley, 2006] ou Direct3D [Microsoft DirectX Specification] que usam a GPU como um coprocessador. As APIs enviam comandos, programas e dados à GPU por meio de um driver de dispositivo gráfico otimizado para a GPU em particular.

Pipeline lógico gráfico

O pipeline lógico gráfico é descrito na Seção A.3. A [Figura A.2.3](#) ilustra os principais estágios de processamento, destacando os estágios programáveis importantes (estágios de sombreamento de vértice, geometria e pixel).

Mapeando o pipeline gráfico a processadores de GPU unificados

A [Figura A.2.4](#) mostra como o pipeline lógico compreendendo estágios programáveis independentes separados é mapeado em um array físico distribuído de processadores.

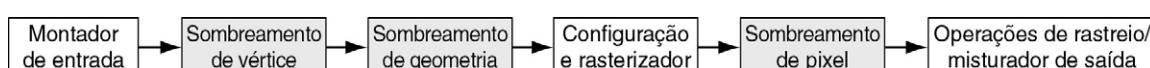


FIGURA A.2.3 Pipeline lógica gráfica. Os estágios de sombreamento programável estão sombreados, e os blocos de função fixa aparecem com fundo branco.

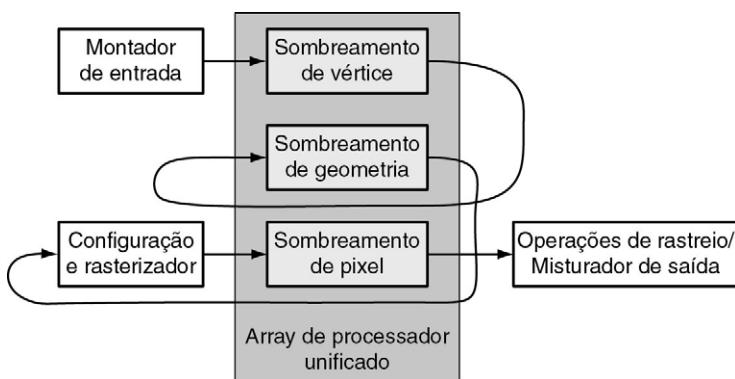


FIGURA A.2.4 Pipeline lógico mapeado nos processadores físicos. Os estágios de sombreamento programável são executados no array dos processadores unificados, e o fluxo de dados do pipeline gráfico lógico recircula pelos processadores.

Arquitetura unificada básica da GPU

As arquiteturas unificadas da GPU são baseadas em um array paralelo de muitos processadores programáveis. Elas unificam o processamento do sombreamento de vértice, geometria e pixel e a computação paralela nos mesmos processadores, diferente das GPUs anteriores, que tinham processadores separados dedicados a cada tipo de processamento. O array de processador programável é altamente integrado com processadores de função fixa para filtragem de textura, rasterização, operações de rastreio, anti-aliasing, compactação, descompactação, exibição, decodificação de vídeo e processamento de vídeo de alta definição. Embora os processadores de função fixa excedam significativamente os processadores programáveis mais genéricos em termos de desempenho absoluto, restringido por um orçamento de área, custo ou potência, vamos focalizar aqui os processadores programáveis.

Em comparação com CPUs multicore, GPUs manycore possuem um objetivo de projeto arquitetônico diferente, focalizado na execução de muitas threads paralelas eficientemente em muitos cores de processador. Usando muitos cores mais simples e otimizando para o comportamento paralelo de dados entre os grupos de threads, mais do orçamento de transistores por chip é dedicado à computação, e menos às caches no chip e overhead.

Array de processadores

Um array de processador GPU unificado contém muitos cores de processador, normalmente organizados em multiprocessadores multithreaded. A Figura A.2.5 mostra uma GPU com um array de 112 cores de processador streaming (SP), organizados como 14 multiprocessadores streaming (SM) multithreaded. Cada core de SP é altamente multithreaded, controlando 96 threads concorrentes e seu estado no hardware. Os processadores se conectam a quatro partições de DRAM com 64 bits de largura por meio de uma rede de interconexão. Cada SM tem oito cores SP, duas unidades de função especial (SFUs), caches de instrução e constante, uma unidade de instrução multithreaded e uma memória compartilhada. Essa é a arquitetura Tesla básica implementada pelo NVIDIA GeForce 8800. Ele tem uma arquitetura unificada em que os programas gráficos tradicionais para o sombreamento de vértice, geometria e pixel executam nas SMs unificadas e seus cores SP, e programas de cálculo executam nos mesmos processadores.

A arquitetura de array do processador é escalável para configurações de GPU menores e maiores, escalando o número de multiprocessadores e o número de partições de memória. A Figura A.2.5 mostra sete clusters de dois SMs compartilhando uma unidade de textura e uma cache L1 de textura. A unidade de textura gera resultados filtrados ao SM dado um conjunto de coordenadas em um mapa de textura. Como as regiões de filtro do suporte normalmente se sobrepõem para solicitações de textura sucessivas, uma pequena cache de textura L1 de streaming é eficaz para reduzir o número de solicitações ao sistema de memória. O array de processadores se conectam com processadores de operação de

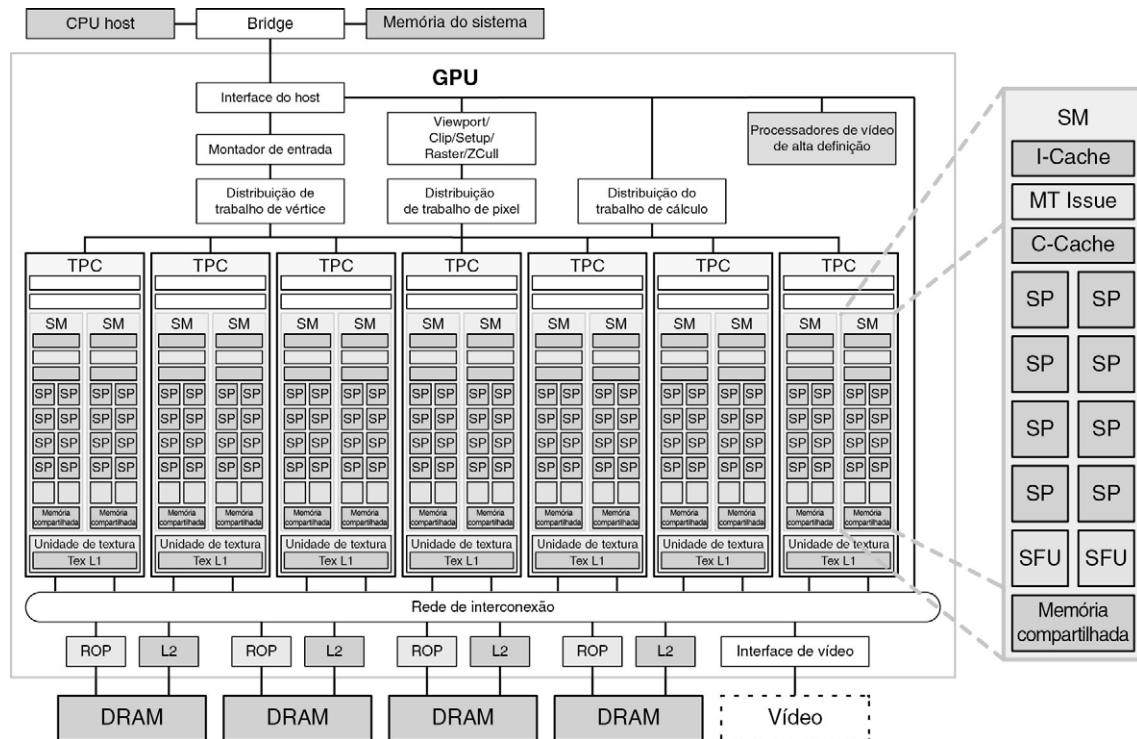


FIGURA A.2.5 Arquitetura unificada básica da GPU. GPU de exemplo com 112 cores de processador streaming (SP) organizados em 14 multiprocessadores streaming (SMs); os cores são altamente multithreaded. Ela tem a arquitetura Tesla básica de um NVIDIA GeForce 8800. Os processadores se conectam com quatro partições DRAM com 64 bits de largura por meio de uma rede de interconexão. Cada SM tem oito cores SP, duas unidades de função especial (SFUs), caches de instrução e constante, uma unidade de instrução multithreaded e uma memória compartilhada.

rastreamento (ROP), caches de textura L2, memórias DRAM externas e memória do sistema por meio de uma rede de interconexão da largura da GPU. O número de processadores e o número de memórias podem se expandir para projetar sistemas de GPU平衡ados para diferentes segmentos de desempenho e mercado.

A.3

Programando GPUs

Programar GPUs de multiprocessador é qualitativamente diferente de programar outros multiprocessadores, como CPUs multicore. As GPUs oferecem duas ou três ordens de grandeza de paralelismo de thread e dados que as CPUs, chegando a centenas de cores de processador e dezenas de milhares de threads simultâneos em 2008. As GPUs continuam a aumentar seu paralelismo, dobrando-o aproximadamente a cada 12 a 18 meses, segundo a lei de Moore [1965] de aumento da densidade do circuito integrado e pela melhoria na eficiência arquitetônica. Para transpor a grande faixa de preço e desempenho dos diferentes segmentos de mercado, diferentes produtos de GPU implementam quantidades bastante variadas de processadores e threads. Mesmo assim, os usuários esperam que aplicações de jogos, gráficos, imagens e cálculo funcionem em qualquer GPU, independente de quantas threads paralelas ela executa ou quantos cores de processador paralelos ela tenha, e eles esperam que GPUs mais caras (com mais threads e cores) rodem as aplicações mais rapidamente. Como resultado, os modelos de programação de GPU e os programas de aplicação são projetados para se expandirem transparentemente a uma grande extensão de paralelismo.

A força motriz por trás do grande número de threads e cores paralelos em uma GPU é o desempenho gráfico em tempo real – a necessidade de renderizar cenas 3D complexas

com alta resolução em taxas de frames interativas, a pelo menos 60 frames por segundo. De modo correspondente, os modelos de programação escaláveis das linguagens de sombreamento gráfico, como Cg (C para gráficos) e HLSL (High-Level Shading Language), são projetados para explorar grandes graus de paralelismo por meio de muitas threads paralelas independentes e expandir para qualquer número de cores de processador. O modelo de programação paralela escalável CUDA, de modo semelhante, permite que aplicações de computação paralela em geral aproveitem grandes números de threads paralelas e se expandam para qualquer número de cores de processador paralelos, transparentemente à aplicação.

Nesses modelos de programação escaláveis, o programador escreve código para uma única thread, e a GPU executa milhares de instâncias de thread em paralelo. Os programas, assim, se expandem transparentemente por uma grande faixa de paralelismo de hardware. Esse paradigma simples surgiu a partir das APIs gráficas e linguagens de sombreamento que descrevem como sombrear um vértice ou um pixel. Ele continuou sendo um paradigma eficaz à medida que as GPUs rapidamente aumentaram seu paralelismo e desempenho desde o final dos anos 90.

Esta seção descreve rapidamente as GPUs de programação para aplicações gráficas de tempo real usando APIs gráficas e linguagens de programação. Depois ela descreve as GPUs de programação para aplicações de computação visual e cálculo paralelo em geral usando a linguagem C e o modelo de programação CUDA.

Programando gráficos em tempo real

As APIs desempenharam um papel importante no desenvolvimento rápido e bem-sucedido das GPUs e processadores. Existem duas APIs gráficas padrão principais: **OpenGL** e **Direct3D**, uma das interfaces de programação de multimídia DirectX da Microsoft. OpenGL, um padrão aberto, foi proposto originalmente e definido pela Silicon Graphics Incorporated. O desenvolvimento e extensão contínua do padrão OpenGL ([Segal e Akeley, 2006], [Kessenich, 2006]) é algo gerenciado pela Khronos, um consórcio do setor. Direct3D [Blythe, 2006], um padrão de fato, é definido e desenvolvido ainda mais pela Microsoft e seus parceiros. OpenGL e Direct3D são estruturados de modo semelhante, e continuam a se desenvolver rapidamente com os avanços do hardware de GPU. Eles definem um pipeline de processamento gráfico lógico que é mapeado no hardware de GPU e processadores, juntamente com modelos de programação e linguagens para os estágios de pipeline programáveis.

OpenGL Uma API gráfica de padrão aberto.

Direct3D Uma API gráfica definida pela Microsoft e seus parceiros.

Pipeline gráfico lógico

A [Figura A.3.1](#) ilustra o pipeline gráfico lógico Direct3D 10. OpenGL tem uma estrutura de pipeline gráfico semelhante. A API e pipeline lógico oferecem uma infraestrutura de

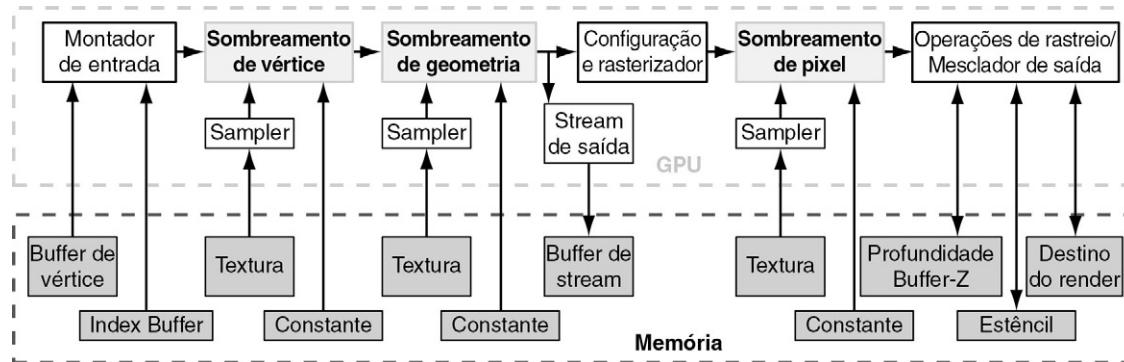


FIGURA A.3.1 Pipeline gráfico Direct3D 10. Cada estágio de pipeline gráfico é mapeado no hardware de GPU ou em um processador de GPU. Os estágios de sombreamento programável estão em azul, os blocos de função fixa estão em branco e os objetos da memória estão em cinza. Cada estágio processa um vértice, primitivo geométrico ou pixel em um padrão de fluxo de dados de streaming.

fluxo de dados de streaming e canalização para os estágios de sombreamento programável, mostrados em azul. A aplicação 3D envia à GPU uma sequência de vértices agrupados em primitivos geométricos – pontos, linhas, triângulos e polígonos. O montador de entrada coleta vértices e primitivos. O programa de sombreamento de vértice executa o processamento por cada vértice, incluindo a transformação da posição 3D do vértice em uma posição de tela e acendendo o vértice para determinar sua cor. O programa de sombreamento de geometria executa o processamento por cada primitivo e pode adicionar ou remover primitivos. A unidade de configuração e o rasterizador geram fragmentos de pixel (fragmentos são contribuições em potencial aos pixels) que são cobertos por um primitivo geométrico. O programa de sombreamento de pixel realiza o processamento por fragmento, incluindo a interpolação de parâmetros por fragmento, texturização e coloração. Os sombreamentos de pixel utilizam bastante as pesquisas amostradas e filtradas em grandes arrays 1D, 2D ou 3D, chamadas **texturas**, usando coordenadas de ponto flutuante interpoladas. Sombreamentos utilizam acessos à textura para mapas, funções, decalques, imagens e dados. O estágio de processamento de operações de rastreio (ou misturador de saída) realiza teste de profundidade do buffer Z e teste de estêncil, que pode descartar um fragmento de pixel oculto ou substituir a profundidade do pixel pela profundidade do fragmento, e realiza uma operação de combinação de cores, que combina a cor do fragmento com a cor do pixel e escreve o pixel com a cor combinada.

A API gráfica e o pipeline gráfico oferecem objetos de entrada, saída, memória e infraestrutura para os programas de sombreamento que processam cada fragmento de vértice, primitivo ou pixel.

textura Um array 1D, 2D ou 3D que admite pesquisas amostradas e filtradas com coordenadas interpoladas.

sombreamento Um programa que opera sobre dados gráficos, como um vértice ou um fragmento de pixel.

linguagem de shading Uma linguagem de renderização de gráficos, normalmente com um modelo de programação de fluxo de dados ou streaming.

Programas de sombreamento gráfico

Aplicações gráficas de tempo real utilizam muitos programas de **sombreamento** diferentes para modelar o modo como a luz interage com diferentes materiais e renderizar iluminação e sombras complexas. As **linguagens de shading** (ou sombreado) são baseadas em um modelo de programação de fluxo de dados ou streaming, que corresponde ao pipeline gráfico lógico. Os programas de sombreamento de vértice mapeiam a posição dos vértices do triângulo na tela, alterando sua posição, cor ou orientação. Normalmente, uma thread do sombreamento de vértice entra com uma posição de vértice em ponto flutuante (x, y, z, w) e calcula uma posição de tela em ponto flutuante (x, y, z). Os programas de sombreamento de geometria operam sobre primitivos geométricos (como linhas e triângulos) definidos por múltiplos vértices, alterando-os ou gerando primitivos adicionais. Os sombreamentos de fragmento de pixel “sobreiam” um pixel cada, calculando uma contribuição de cor vermelho, verde, azul, alfa (RGBA) de ponto flutuante para a imagem renderizada em sua posição de imagem (x, y) da amostra do pixel. Sombreamentos (e GPUs) utilizam aritmética de ponto flutuante para todos os cálculos de cor de pixel, a fim de eliminar artefatos visíveis enquanto calcula a faixa extrema de valores de contribuição de pixel encontrados na renderização de cenas com iluminação complexa, sombras e alta faixa dinâmica. Para todos os três tipos de sombreamentos gráficos, muitas instâncias de programa podem ser executadas em paralelo, como threads paralelas independentes, pois cada uma trabalha sobre dados independentes, produz resultados independentes e não tem efeitos colaterais. Vértices, primitivos e pixels independentes ainda permitem que o mesmo programa gráfico seja executado em GPUs de tamanhos diferentes, que processam diferentes quantidades de vértices, primitivos e pixels em paralelo. Assim, programas gráficos se expandem transparentemente às GPUs com diferentes quantidades de paralelismo e desempenho.

Os usuários programam todas as três threads gráficas com uma linguagem de alto nível dirigida comum. HLSL (High-Level Shading Language) e Cg (C para gráficos) normalmente são usadas. Elas possuem sintaxe tipo C e um rico conjunto de funções de biblioteca para operações de matriz, trigonometria, interpolação e acesso e filtragem de textura, mas estão longe de ser linguagens de computação de uso geral: atualmente não possuem acesso à memória geral, ponteiros, E/S de arquivo e recursão. HLSL e Cg consideram que os programas residem dentro de um pipeline gráfico lógico, e, portanto, a E/S é implícita.

Por exemplo, um sombreamento de fragmento de pixel pode esperar que as coordenadas de texto geométrica normal e múltipla tenham sido interpoladas a partir dos valores de vértice por estágios de função fixa anteriores, e pode simplesmente atribuir um valor ao parâmetro de saída COLOR para que seja passado adiante, de modo a ser misturado com um pixel em uma posição (x, y) implícita.

O hardware da GPU cria uma nova thread independente para executar um programa de sombreamento de vértice, geometria ou pixel para cada vértice, cada primitivo e cada fragmento de pixel. Nos *video games*, a maior parte dos threads executa programas de sombreamento de pixel, pois normalmente existem de 10 a 20 vezes ou mais fragmentos de pixel do que vértices, e iluminação e sombras complexas exigem razões ainda maiores entre threads de sombreamento de pixel e vértice. O modelo de programação de sombreamento gráfico impulsionou a arquitetura de GPU a executar de modo eficiente milhares de threads fine-grained independentes em muitos cores paralelos do processador.

Exemplo de sombreamento de pixel

Considere o seguinte programa sombreamento de pixel Cg, que implementa a técnica de renderização de “mapeamento de ambiente”. Para cada thread de pixel, esse sombreamento recebe cinco parâmetros, incluindo coordenadas da imagem de textura em ponto flutuante 2D, necessárias para amostrar a cor da superfície, e um vetor de ponto flutuante 3D dando a reflexão da direção da visão a partir da superfície. Os outros três parâmetros “uniformes” não variam de uma instância de pixel (thread) para a seguinte. O sombreamento pesquisa a cor nas duas imagens de textura: um acesso de textura 2D para a cor da superfície e um acesso de textura 3D em um mapa de cubo (seis imagens correspondentes às faces de um cubo) para obter a cor do mundo exterior correspondente à direção de reflexão. Depois, a cor de ponto flutuante final com quatro componentes (vermelho, verde, azul, alfa) é calculada usando uma média ponderada chamada “lerp”, ou função de interpolação linear.

```
void reflection(
    float2          texCoord      : TEXCOORD0,
    float3          reflection_dir : TEXCOORD1,
    out float4      color         : COLOR,
    uniform float   shiny,
    uniform sampler2D surfaceMap,
    uniform samplerCUBE envMap)
{
    // Apanha a cor de superfície a partir de uma textura
    float4 surfaceCol = tex2D(surfaceMap, texCoord);

    // Apanha a cor refletida amostrando um mapa de cubo
    float4 reflectEdCol = texCUBE(environmentMap, reflection_dir);

    // A saída é a média ponderada das duas cores
    color = lerp(surfaceColor, reflectEdColor, shiny);
}
```

Embora esse programa de sombreamento tenha apenas três linhas, ele ativa muito hardware da GPU. Para cada busca de textura, o subsistema de textura da GPU faz diversos acessos à memória para amostrar cores da imagem nas vizinhanças das coordenadas de amostragem, e depois interpola o resultado final com a aritmética de filtragem em ponto flutuante. A GPU multithreaded executa milhares dessas threads peso leve de sombreamento de pixel Cg em paralelo, intercalando-as profundamente para ocultar a busca de textura e latência de memória.

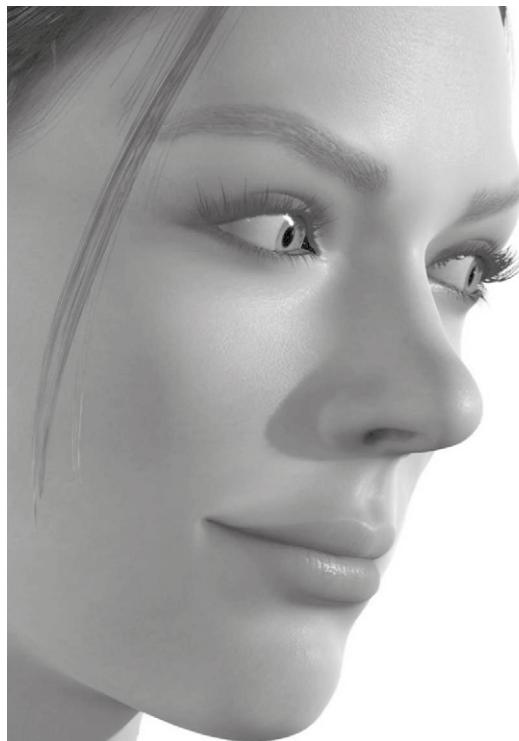


FIGURA A.3.2 Imagem renderizada pela GPU. Para que a pele tenha profundidade e translucidez visual, o programa de sombreamento de pixel modela três camadas de pele separadas, cada uma com comportamento exclusivo de espalhamento da subsuperfície. Ele executa 1400 instruções para renderizar os componentes de cor vermelho, verde, azul e alfa de cada fragmento de pixel da pele.

Cg focaliza a visão do programador de um único vértice ou primitivo ou pixel, que a GPU implementa como uma única thread; o programa de sombreamento é dimensionado transparentemente para explorar o paralelismo de thread nos processadores disponíveis. Sendo específica da aplicação, Cg oferece um rico conjunto de tipos de dados úteis, funções de biblioteca e construções de linguagem para expressar técnicas de renderização variadas.

A Figura A.3.2 mostra a pele renderizada por um sombreamento de pixel de fragmento. A pele real parece muito diferente da tinta de cor da carne, pois a luz salta muito antes de emergir novamente. Nesse sombreamento complexo, três camadas de pele separadas, cada uma com comportamento de espalhamento de subsuperfície exclusivo, são modeladas para oferecer à pele profundidade e translucidez visual. O espalhamento pode ser modelado por uma convolução ofuscante em um espaço de “textura” aplaudido, com o vermelho sendo ofuscado mais que o verde, e o azul ofuscado menos. O sombreamento Cg compilado executa 1400 instruções para calcular a cor de um pixel de pele.

Como as GPUs desenvolveram um desempenho de ponto flutuante superior e largura de banda de memória streaming muito alta para gráficos em tempo real, elas atraíram aplicações altamente paralelas além dos gráficos tradicionais. Inicialmente, o acesso a esse poder estava disponível apenas formulando uma aplicação como um algoritmo de renderização de gráficos, mas essa técnica GPGPU normalmente era esquisita e limitadora. Mais recentemente, o modelo de programação CUDA forneceu um modo muito mais fácil de explorar a largura de banda escalável de ponto flutuante e memória de alto desempenho das GPUs com a linguagem de programação C.

Programando aplicações de computação paralela

CUDA, Brook e CAL são interfaces de programação para GPUs que são focadas na computação paralela dos dados, em vez de gráficos. CAL (Compute Abstraction Layer) é uma interface de linguagem assembler de baixo nível para GPUs da AMD. Brook é uma

linguagem de streaming adaptada para BPUs por Buck *et al.* [2004]. CUDA, desenvolvida pela NVIDIA [2007], é uma extensão às linguagens C e C++ para a programação paralela escalável de GPUs manycore e CPUs multicore. O modelo de programação CUDA é descrito a seguir, adaptado de um artigo de Nickolls, Buck, Garland e Skadron [2008].

Com o novo modelo, a GPU excede em cálculo paralelo de dados e vazão, executando aplicações de cálculo de alto desempenho além de aplicações gráficas.

Decomposição do problema paralelo de dados

Para mapear grandes problemas de cálculo de modo eficaz a uma arquitetura de processamento altamente paralela, o programador ou compilador decompõe o problema em muitos problemas pequenos, que podem ser solucionados em paralelo. Por exemplo, o programador divide um array de dados de resultado grande em blocos e divide ainda mais cada bloco em elementos, de modo que os blocos de resultado possam ser calculados independentemente em paralelo, e os elementos dentro de cada bloco sejam calculados em paralelo. A [Figura A.3.3](#) mostra uma decomposição de um array de dados de resultado em uma grande de blocos 3×2 , em que cada bloco é decomposto ainda mais em um array 5×3 de elementos. A decomposição paralela em dois níveis é mapeada naturalmente para a arquitetura da GPU: multiprocessadores paralelos calculam blocos de resultado, e threads paralelas calculam elementos de resultado.

O programador escreve um programa que calcula uma sequência de grades de dados de resultado, dividindo cada grade de resultado em blocos de resultado coarse-grained, que podem ser calculados independentemente em paralelo. O programa calcula cada bloco de resultado com um array de threads paralelas fine-grained, particionando o trabalho entre threads, de modo que cada uma calcule um ou mais elementos de resultado.

Programação paralela escalável com CUDA

O modelo de programação paralela escalável CUDA estende as linguagens C e C++ para explorar grandes graus de paralelismo para aplicações gerais em multiprocessadores altamente paralelos, particularmente GPUs. A experiência inicial com CUDA mostra que muitos programas sofisticados podem ser prontamente expressos com algumas

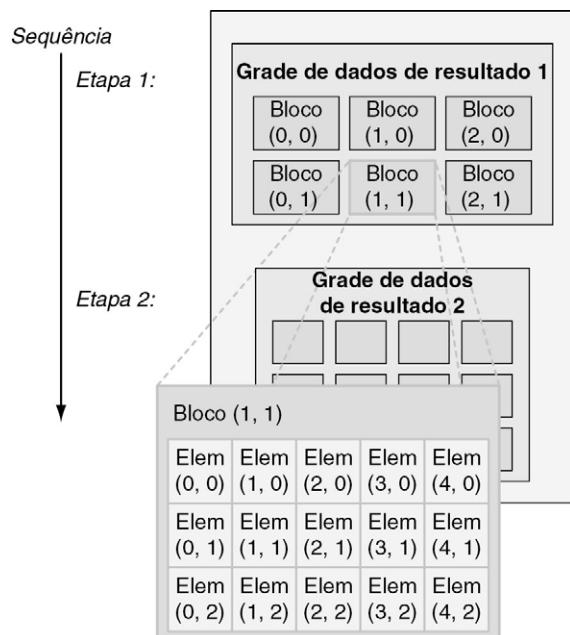


FIGURA A.3.3 Decompondo os dados de resultado em uma grade de blocos de elementos a serem calculados em paralelo.

abstrações facilmente entendidas. Como a NVIDIA lançou o modelo CUDA em 2007, os desenvolvedores rapidamente criaram programas paralelos escaláveis para uma grande faixa de aplicações, incluindo processamento de dados sísmicos, química computacional, álgebra linear, solucionadores de matriz esparsa, classificação, pesquisa, modelos da física e computação visual. Essas aplicações se expandem transparentemente para centenas de cores de processador e milhares de threads simultâneas. As GPUs NVIDIA com a arquitetura gráfica e de computação unificada Tesla (descrita nas Seções A.4 e A.7) executam programas CUDA em C, e se encontram facilmente em laptops, PCs, estações de trabalho e servidores. O modelo CUDA também se aplica a outras arquiteturas de processamento paralelo de memória compartilhada, incluindo CPUs multicore [Stratton, 2008].

CUDA oferece três abstrações principais – uma *hierarquia de grupos de threads, memórias compartilhadas e sincronismo de barreira* –, que oferecem uma estrutura paralela clara ao código C convencional para uma thread da hierarquia. Múltiplos níveis de threads, memória e sincronização oferecem paralelismo de dados fine-grained e paralelismo de threads, aninhados dentro do paralelismo de dados coarse-grained e paralelismo de tarefas. As abstrações orientam o programador a dividir o problema em subproblemas grosseiros, que podem ser solucionados independentemente em paralelo, e depois em partes mais minuciosas, que podem ser solucionadas em paralelo. O modelo de programação se expande transparentemente para quantidades maiores de cores de processador: um programa CUDA compilado é executado em qualquer número de processadores, e somente o sistema de runtime precisa saber a quantidade física de processadores.

O paradigma CUDA

kernel Um programa ou função para uma thread, projetado para ser executado por muitas threads.

bloco de threads Um conjunto de threads simultâneas, que executam o mesmo programa de thread e podem cooperar para calcular um resultado.

grade Um conjunto de blocos de threads que executam o mesmo programa do kernel.

CUDA é uma extensão mínima das linguagens de programação C e C++. O programador escreve um programa serial que chama **kernels** paralelos, que podem ser funções simples ou programas completos. Um kernel é executado em paralelo a um conjunto de threads paralelas. O programador organiza essas threads em uma hierarquia de blocos de threads e grades de blocos de threads. Um **bloco de threads** é um conjunto de threads simultâneas que podem cooperar entre si através da sincronização de barreira e através do acesso compartilhado ao espaço de memória privado do bloco. Uma **grade** é um conjunto de blocos de threads que podem ser executadas independentemente e, portanto, podem ser executadas em paralelo.

Ao chamar um kernel, o programador especifica o número de threads por bloco e o número de blocos que compreendem a grade. Cada grade recebe um número de *thread ID* exclusivo, *threadIdx*, dentro do seu bloco de threads, numerado com 0, 1, 2, ..., *blockDim*-1, e cada bloco de threads recebe um número de *block ID* exclusivo, *blockIdx*, dentro de sua grade. CUDA admite blocos de threads contendo até 512 threads. Por conveniência, os blocos de threads e grades podem ter uma, duas ou três dimensões, acessadas por meio dos campos de índice .x, .y e .z.

Como um exemplo muito simples de programação paralela, suponha que recebemos dois vetores *x* e *y* de *n* números de ponto flutuante cada, e que queiramos calcular o resultado de $y = ax + y$ para algum valor escalar *a*. Esse é o chamado kernel SAXPY, definido pela bib de álgebra linear BLAS. A Figura A.3.4 mostra o código C para realizar esse cálculo em um processador serial e em paralelo, usando CUDA.

O especificador de declaração `_global_` indica que o procedimento é um ponto de entrada do kernel. Os programas em CUDA disparam kernels paralelos com a sintaxe estendida de chamada de função:

```
kernel <<< dimGrid, dimBlock >>> (...lista de parâmetros ...);
```

onde `dimGrid` e `dimBlock` são vetores de três elementos do tipo `dim3`, que especificam as dimensões da grade em blocos e as dimensões dos blocos em threads, respectivamente. Dimensões não especificadas são um por default.

Calculando $y = ax + y$ com um loop serial:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Chama kernel SAXPY serial
saxpy_serial(n, 2.0, x, y);
```

Calculando $y = ax + y$ em paralelo usando CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n ) y[i] = alpha*x[i] + y[i];
}

// Chama kernel SAXPY paralelo (256 threads por bloco)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

FIGURA A.3.4 Código sequencial (em cima) em C versus código paralelo (embaixo) em CUDA para SAXPY (veja Capítulo 7). Threads paralelas CUDA substituem o loop serial C — cada thread calcula o mesmo resultado como uma iteração do loop. O código paralelo calcula n resultados com n threads organizados em blocos de 256 threads.

Na Figura A.3.4, disparamos uma grade de n threads que atribui uma thread a cada elemento dos vetores e coloca 256 threads em cada bloco. Cada thread individual calcula um índice de elemento de sua thread e IDs de bloco, e depois realiza o cálculo desejado nos elementos de vetor correspondentes. Comparando as versões serial e paralelo desse código, vemos que elas são muito semelhantes. Isso representa um padrão bastante comum. O código serial consiste em um loop em que cada iteração é independente de todas as outras. Esses loops podem ser transformados matematicamente em kernels paralelos: cada iteração do loop se torna uma thread independente. Atribuindo uma única thread a cada elemento de saída, evitamos a necessidade de qualquer sincronização entre as threads ao escrever resultados na memória.

O texto de um kernel CUDA é simplesmente uma função C para uma thread sequencial. Assim, ele geralmente é fácil de escrever e mais simples do que escrever código paralelo para operações de vetor. O paralelismo é determinado clara e explicitamente especificando-se as dimensões de uma grade e seus blocos de threads ao iniciar um kernel.

A execução paralela e o gerenciamento de threads é automático. Toda criação, escalonamento e término de thread é tratado para o programador pelo sistema subjacente. Na verdade, uma GPU com arquitetura Tesla realiza todo o gerenciamento de threads diretamente no hardware. As threads de um bloco são executadas simultaneamente e podem sincronizar em uma **barreira de sincronização** chamando o `_syncthreads()` intrínseco. Isso garante que nenhuma thread no bloco possa prosseguir até que todas as threads em um bloco tenham chegado à barreira. Depois de passar pela barreira, essas threads também têm garantias de ver todas as escritas na memória realizadas pelas threads no bloco antes da barreira. Assim, as threads em um bloco podem se comunicar entre si escrevendo e lendo a memória compartilhada por bloco em uma barreira de sincronização.

Como as threads em um bloco podem compartilhar memória e sincronizar por meio de barreiras, elas residirão juntas no mesmo processador ou multiprocessador físico. Porém,

barreira de sincronização

Threads esperam em uma barreira de sincronização até que todas as threads no bloco de threads cheguem à barreira.

o número de blocos de threads pode ser muito superior ao número de processadores. O modelo de programação de threads CUDA virtualiza os processadores e dá ao programador a flexibilidade de paralelizar em qualquer granularidade que seja mais conveniente. A virtualização em threads e blocos de threads permite decomposições intuitivas do problema, pois o número de blocos pode ser ditado pelo tamanho dos dados sendo processados, em vez do número de processadores no sistema. Ele também permite que o mesmo programa CUDA se expanda para números bastante variáveis de cores de processador.

Para gerenciar essa virtualização do elemento de processamento e oferecer escalabilidade, o CUDA requer que os blocos de threads possam ser executados independentemente. Deverá ser possível executar os blocos em qualquer ordem, em paralelo ou em série. Diferentes blocos não têm meios de comunicação direta, embora possam *coordenar* suas atividades usando **operações de memória atômicas** na memória global visível a todas as threads – incrementando atomicamente os ponteiros de fila, por exemplo. Esse requisito de independência permite que os blocos de threads sejam escalonados em qualquer ordem por qualquer número de cores, tornando o modelo CUDA expansível por um número qualquer de cores, bem como por uma série de arquiteturas paralelas. Ele também ajuda a evitar a possibilidade de deadlock. Uma aplicação pode executar múltiplas grades de forma dependente ou independente. Grades independentes podem ser executadas simultaneamente, dados recursos de hardware suficientes. Grades dependentes são executadas sequencialmente, com uma barreira implícita entre os kernels, garantindo assim que todos os blocos da primeira grade terminem antes do início de qualquer bloco da segunda grade, dependente.

As threads podem acessar dados de vários espaços da memória durante sua execução. Cada thread tem uma **memória local** particular. O modelo CUDA utiliza a memória local para variáveis particulares da thread, que não se encaixam nos registradores da thread, além de frames de pilha e derramamento de registrador. Cada bloco de thread tem uma **memória compartilhada**, visível a todas as threads do bloco, que tem o mesmo tempo de vida do bloco. Finalmente, todas as threads têm acesso à mesma **memória global**. Os programas declaram variáveis na memória compartilhada e global com os qualificadores de tipo `_shared_` e `_device_`. Em uma GPU na arquitetura Tesla, esses espaços de memória correspondem a memórias fisicamente separadas: a memória compartilhada por bloco é uma RAM no chip com baixa latência, enquanto a memória global reside na DRAM rápida da placa gráfica.

A memória compartilhada deverá ser uma memória de baixa latência perto de cada processador, semelhante a uma cache L1. Portanto, ela pode oferecer comunicação de alto desempenho e compartilhamento de dados entre as threads de um bloco de threads. Por ter o mesmo tempo de vida do seu bloco de threads correspondente, o código do kernel normalmente inicializará os dados nas variáveis compartilhadas, calculará usando variáveis compartilhadas e copiará os resultados da memória compartilhada para a memória global. Os blocos de threads de grades sequencialmente dependentes se comunicam por meio da memória global, usando-a para ler entrada e escrever resultados.

A Figura A.3.5 diagrama os níveis aninhados das threads, blocos de threads e grades dos blocos de threads. Ela mostra ainda os níveis correspondentes de compartilhamento de memória: memórias locais, compartilhadas e globais para o compartilhamento de dados por thread, por bloco de threads e por aplicação.

Um programa gerencia o espaço da memória global visível aos kernels por meio de chamadas ao runtime CUDA, como `cudaMalloc()` e `cudaFree()`. Os kernels podem ser executados em um dispositivo fisicamente separado, como é o caso quando os kernels são executados na GPU. Consequentemente, a aplicação precisa usar `cudaMemcpy()` para copiar dados entre o espaço alocado e a memória do sistema host.

O modelo de programação CUDA é semelhante em estilo ao conhecido modelo *single-program multiple data (SPMD)* — ele expressa paralelismo explicitamente, e cada kernel executa em um número fixo de threads. Porém, CUDA é mais flexível que a maioria das realizações do modelo SPMD, pois cada chamada do kernel cria dinamicamente uma nova grade com o número correto de blocos de thread e threads para essa etapa da aplicação. O

operação de memória atômica

atômica Uma sequência de operações de leitura, modificação e escrita de memória que termina sem qualquer acesso intervente.

memória local Memória local por thread, privativa à thread.

memória compartilhada Memória por bloco, compartilhada por todas as threads do bloco.

memória global Memória por aplicação, compartilhada por todas as threads.

Single-Program Multiple Data (SPMD)

Um estilo de modelo de programação paralela em que todas as threads executam o mesmo programa. Threads SPMD normalmente são coordenados com sincronização de barreira.

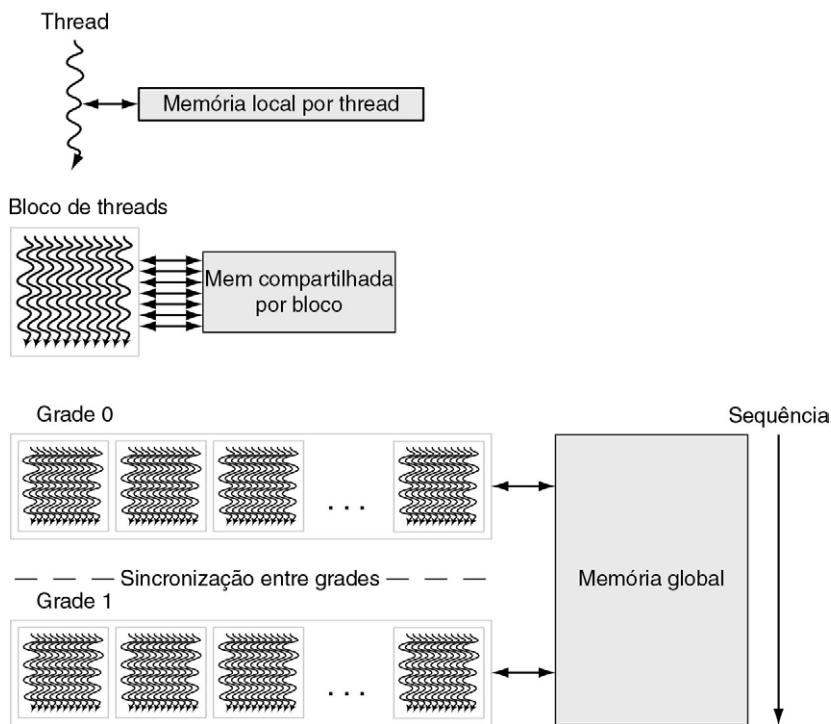


FIGURA A.3.5 Níveis de granularidade aninhados — thread, bloco de threads e grade — possuem níveis de compartilhamento de memória correspondentes — local, compartilhado e global. A memória local por thread é particular à thread. A memória compartilhada por bloco é compartilhada por todas as threads do bloco. A memória global por aplicação é compartilhada por todas as threads.

programador pode usar um grau de paralelismo conveniente para cada kernel, em vez de ter de projetar todas as fases do cálculo para usar o mesmo número de threads. A Figura A.3.6 mostra um exemplo de uma sequência de código CUDA tipo SPMD. Primeiro, o código instancia o kernel F em uma grade 2D de 3×2 blocos, em que cada bloco de threads 2D consiste em 5×3 threads. Depois, ele instancia kernel G em uma grade 1D de quatro blocos de threads 1D com seis threads cada. Como kernel G depende dos resultados de kernel F, eles são separados por uma barreira de sincronização entre kernels.

As threads simultâneas de um bloco de threads expressam paralelismo de dados fine-grained e paralelismo de thread. Os blocos de threads independentes de uma grade expressam paralelismo de dados coarse-grained. Grades independentes expressam o paralelismo de tarefas coarse-grained. Um kernel é simplesmente código C para uma thread da hierarquia.

Restrições

Por eficiência, e para simplificar sua implementação, o modelo de programação CUDA tem algumas restrições. Threads e blocos de threads só podem ser criados chamando-se um kernel paralelo, não de dentro de um kernel paralelo. Junto com a independência exigida dos blocos de threads, isso possibilita executar programas CUDA com um scheduler simples, que introduz um overhead mínimo em runtime. De fato, a arquitetura GPU Tesla implementa gerenciamento de *hardware* e escalonamento de threads e blocos de threads.

O paralelismo de tarefa pode ser expresso no nível de bloco de threads, mas é difícil de expressar dentro de um bloco de threads, pois as barreiras de sincronização de thread operam sobre todas as threads do bloco. Para permitir que os programas CUDA sejam executados em qualquer número de processadores, as dependências entre os blocos de threads dentro da mesma grade de kernel não são permitidas — os blocos precisam ser executados independentemente. Como CUDA exige que os blocos de threads sejam independentes e

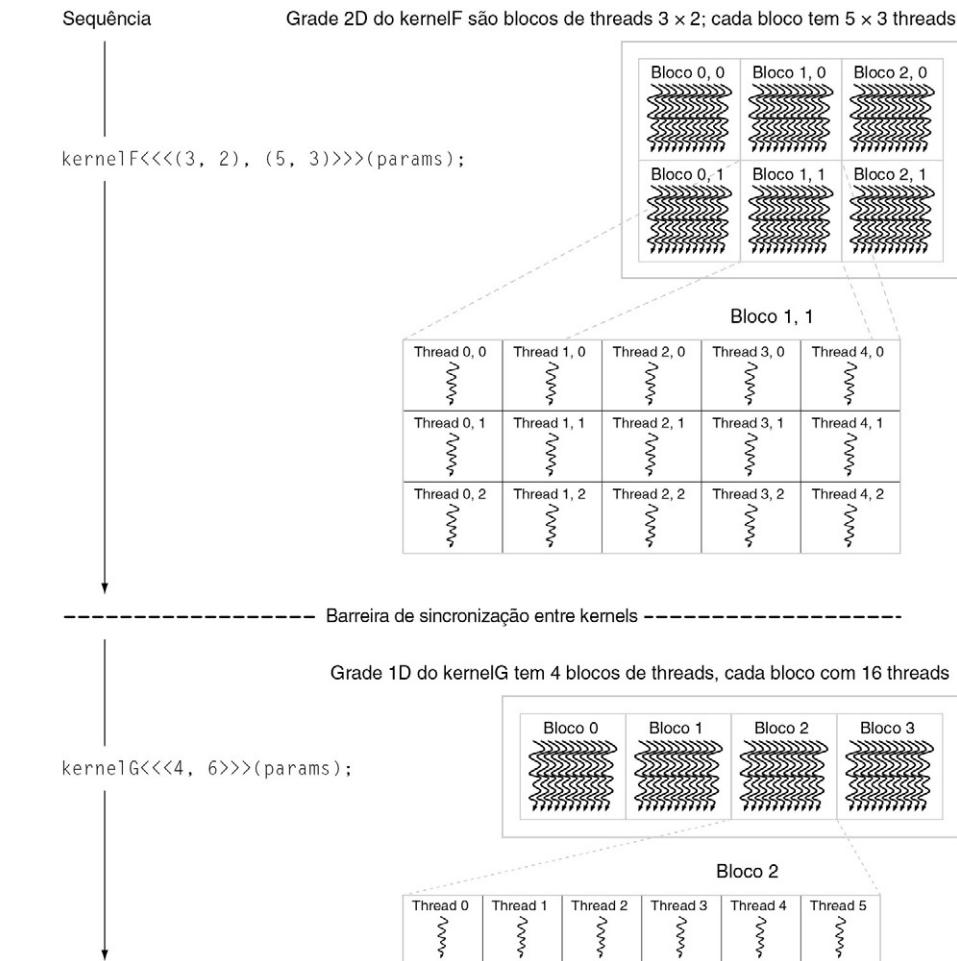


FIGURA A.3.6 Sequência de kernel F instanciado em uma grade 2D de blocos de threads 2D, uma barreira de sincronização entre kernels, seguida por kernel G em uma grade 1D de blocos de threads 1D.

permite que os blocos sejam executados em qualquer ordem, a combinação dos resultados gerados por múltiplos blocos em geral precisa ser feita iniciando um segundo kernel em uma nova grade de blocos de threads (embora os blocos de threads possam *coordenar* suas atividades usando operações de memória atômicas na memória global visível a todas as threads — incrementando atomicamente os ponteiros de fila, por exemplo).

Chamadas de função recursivas atualmente não são permitidas em kernels CUDA. A recursão é pouco atraente em um kernel maciçamente paralelo, pois oferecer espaço de pilha para as dezenas de milhares de threads que podem estar ativas exigiria quantidades substanciais de memória. Algoritmos seriais que normalmente são expressos usando recursão, como o quicksort, normalmente são implementados melhor usando paralelismo de dados aninhado em vez da recursão explícita.

Para dar suporte a uma arquitetura de sistema heterogênea combinando uma CPU e uma GPU, cada uma com seu próprio sistema de memória, os programas CUDA precisam copiar dados e resultados entre a memória do host e a memória do dispositivo. O overhead da interação CPU-GPU e transferências de dados é minimizado usando-se mecanismos de transferência em bloco por DMA e interconexões velozes. Problemas com uso intenso de cálculo, grandes o suficiente para precisar de um aumento de desempenho da GPU, amortizam o overhead melhor do que os problemas pequenos.

Implicações para a arquitetura

Os modelos de programação paralela para gráficos e computação têm feito com que a arquitetura da GPU seja diferente da arquitetura da CPU. Os principais aspectos dos programas da GPU impulsionando a arquitetura de processador da GPU são:

- *Uso extenso do paralelismo de dados fine-grained*: Programas de sombreamento descrevem como processar um único pixel ou vértice, e programas CUDA descrevem como calcular um resultado individual.
- *Modelo de programação altamente encadeado*: Um programa de thread de sombreamento processa um único pixel ou vértice, e um programa de thread CUDA pode gerar um único resultado. Uma GPU precisa criar e executar milhões desses programas de thread por frame, a 60 frames por segundo.
- *Escalabilidade*: Um programa precisa aumentar automaticamente seu desempenho quando receber processadores adicionais, sem recompilação.
- *Cálculo intenso de ponto flutuante (ou inteiro)*.
- *Suporte para cálculos com alta vazão*.

A.4

Arquitetura de multiprocessador multithreaded

Para lidar com diferentes segmentos do mercado, as GPUs implementam números escaláveis de multiprocessadores — na verdade, as GPUs são multiprocessadores compostos de multiprocessadores. Além do mais, cada multiprocessador é altamente multithreaded para executar muitas threads de sombreamento de vértice e pixel fine-grained de forma eficiente. Uma GPU básica de qualidade tem dois a quatro multiprocessadores, enquanto a GPU ou plataforma de computação de um entusiasta em jogos tem dezenas deles. Esta seção examina a arquitetura de um multiprocessador multithreaded desse tipo, uma versão simplificada do multiprocessador streaming (SM) Tesla da NVIDIA, descrito na Seção A.7.

Por que usar um multiprocessador, em vez de vários processadores independentes? O paralelismo dentro de cada multiprocessador oferece alto desempenho localizado e admite multithreading extenso para os modelos de programação paralela fine-grained descritos na Seção A.3. As threads individuais de um bloco de threads são executadas juntas dentro de um multiprocessador para compartilhar dados. O projeto de multiprocessador multithreaded que descrevemos aqui tem oito cores de processador escalares em uma arquitetura altamente acoplada, e executa até 512 threads (o SM descrito na Seção A.7 executa até 768 threads). Por questão de eficiência de espaço e potência, o multiprocessador compartilha unidades grandes e complexas entre oito cores de processador, incluindo a cache de instruções, a unidade de instrução multithreaded e a RAM de memória compartilhada.

Multithreading maciço

Processadores GPU são altamente multithreaded para alcançar vários objetivos:

- Cobrir a latência de loads da memória e buscas de textura da DRAM.
- Admitir modelos de programação de sombreamento gráfico paralelo fine-grained.
- Admitir modelos de programação de cálculo paralelo fine-grained.
- Virtualizar os processadores físicos como threads e blocos de threads para oferecer escalabilidade transparente.
- Simplificar o modelo de programação paralelo para escrever um programa serial para uma thread.

A latência da busca de memória e textura podem exigir centenas de clocks de processador, pois as GPUs normalmente têm pequenos caches streaming em vez de grandes caches do conjunto de trabalho, como as CPUs. Uma solicitação de busca geralmente requer uma latência de acesso completa da DRAM mais a latência de interconexão e buffering. O multithreading ajuda a cobrir a latência com computação útil — enquanto uma thread está esperando que um load ou busca de textura termine, o processador pode executar outra thread. Os modelos de programação paralela fine-grained oferecem literalmente milhares de threads independentes que podem manter muitos processadores ocupados, apesar da longa latência de memória vista pelas threads individuais.

Um programa gráfico de sombreamento de vértice ou pixel é um programa para uma única thread que processa um vértice ou um pixel. De modo semelhante, um programa CUDA é um programa em C para uma única thread, que calcula um resultado. Programas gráficos e de computação instanciam muitas threads paralelas para renderizar imagens complexas e calcular grandes arrays de resultado. Para balancear dinamicamente o vértice de deslocamento e cargas de trabalho do thread de sombreamento de pixel, cada multiprocessador executa simultaneamente múltiplos programas de thread diferentes e diferentes tipos de programas de sombreamento.

Para dar suporte ao modelo de programação de vértice, primitivo e pixel independentes das linguagens de sombreamento gráfico e o modelo de programação de única thread do CUDA C/C++, cada thread da GPU tem seus próprios registradores privados, memória por thread particular, contador de programa e estado de execução de thread, e pode executar um caminho de código independente. Para executar de modo eficiente centenas de threads leves simultâneas, o multiprocessador da GPU é multithreaded por hardware — ele controla e executa centenas de threads simultâneas no hardware sem overhead de escalonamento. Threads simultâneas dentro de blocos de threads podem sincronizar em uma barreira com uma única instrução. A criação de threads peso leve, escalonamento de threads com overhead zero, e sincronização de barreira rápida efetivamente dão suporte ao paralelismo bastante fine-grained.

Arquitetura de multiprocessador

Um multiprocessador gráfico e de computação unificado executa programas de sombreamento de vértice, geometria e fragmento de pixel, e programas de computação paralelos. Como mostra a [Figura A.4.1](#), o multiprocessador de exemplo consiste em oito cores de processador escalar (SP), cada um com um grande arquivo de registrador (RF) multithreaded, duas unidades de função especial (SFU), uma unidade de instrução multithreaded, uma cache de instrução, uma cache constante apenas de leitura e uma memória compartilhada.

A memória compartilhada de 16KB mantém buffers de dados gráficos e dados de computação compartilhados. Variáveis CUDA declaradas como `__shared__` residem na memória compartilhada. Para mapear a carga de trabalho da pipeline gráfica lógica através dos múltiplos tempos do multiprocessador, como mostra a Seção A.2, threads de vértice, geometria e pixel possuem buffers de entrada e saída independentes, e as cargas de trabalho chegam e saem independentemente da execução da thread.

Cada core do SP contém unidades aritméticas escalares de inteiros e ponto flutuante que executam a maioria das instruções. O SP é multithreaded por hardware, aceitando até 64 threads. Cada core de SP em pipeline executa uma instrução escalar por thread por clock, que varia de 1,2GHz a 1,6GHz em diferentes produtos de GPU. Cada core SP tem um arquivo de registrador (RF) grande de 1024 registradores de uso geral de 32 bits, divididos entre suas threads atribuídas. Os programas declaram sua demanda de registrador, normalmente 16 a 64 registradores escalares de 32 bits por thread. O SP pode executar simultaneamente muitas threads que usam alguns registradores ou menos threads que usam mais registradores. O compilador otimiza a alocação do registrador para balancear o custo do spilling de registradores *versus* o custo de menos threads. Os programas de sombreamento de pixel normalmente utilizam 16 ou menos registradores, permitindo que cada SP execute até 64 threads de sombreamento de pixel para cobrir

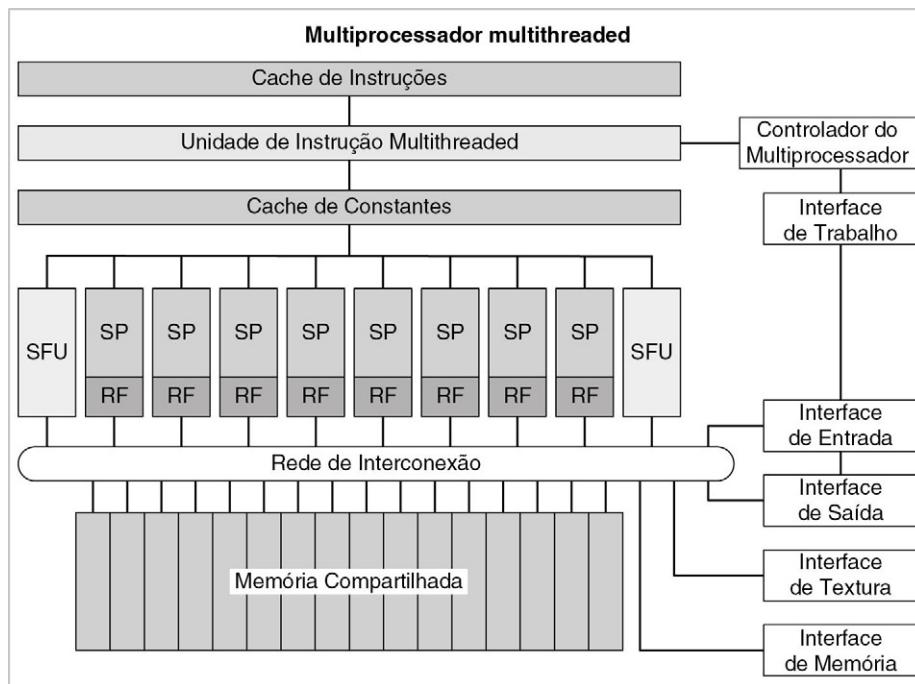


FIGURA A.4.1 Multiprocessador multithreaded com oito cores de processador escalar (SP). Oito cores SP possuem um grande arquivo de registrador (RF) multithreaded cada, e compartilham uma cache de instrução, unidade de emissão de instrução multithreading, cache constante, duas unidades de função especial (SFUs), rede de interconexão e uma memória compartilhada multibanco.

buscas de textura de longa latência. Programas CUDA compilados normalmente precisam de 32 registradores por thread, limitando cada SP a 32 threads, que limita esse programa de kernel a 256 threads por bloco de threads nesse multiprocessador de exemplo, em vez do seu máximo de 512 threads.

As SFUs em pipeline executam instruções de thread que calculam funções especiais e interpolam atributos de pixel dos atributos de vértice primitivos. Essas instruções podem ser executadas simultaneamente com as instruções nos SPs. A SFU é descrita mais adiante.

O multiprocessador executa instruções de busca de textura na unidade de textura por meio da interface de textura, e usa a interface de memória para instruções de load e store da memória externa, e instruções atômicas de acesso. Essas instruções podem executar simultaneamente com instruções nos SPs. O acesso à memória compartilhada usa uma rede de interconexão de baixa latência entre os processadores SP e os bancos de memória compartilhada.

Single-Instruction Multiple-Thread (SIMT)

Para controlar e executar centenas de threads rodando diversos programas diferentes de modo eficaz, o multiprocessador emprega uma arquitetura **Single-Instruction Multiple-Thread (SIMT)**. Ele cria, controla, escalona e executa threads simultâneas em grupos de threads paralelas, chamados *warp*s. O termo *warp* é originado da tecelagem, a primeira tecnologia de thread paralela. A fotografia na Figura A.4.2 mostra um warp de threads paralelas surgindo de um tear. Esse multiprocessador de exemplo utiliza um tamanho de warp SIMT de 32 threads, executando quatro threads em cada um dos oito cores SP por quatro clocks. O multiprocessador Tesla SM descrito na Seção A.7 também usa um tamanho de warp de 32 threads paralelas, executando quatro threads por core SP por eficiência em threads de pixel e threads de cálculo abundantes. Os blocos de threads consistem em um ou mais warps.

Single-Instruction Multiple-Thread (SIMT) Uma arquitetura de processador que aplica uma instrução a múltiplas threads independentes em paralelo.

warp O conjunto de threads paralelas que executam a mesma instrução juntas em uma arquitetura SIMT.

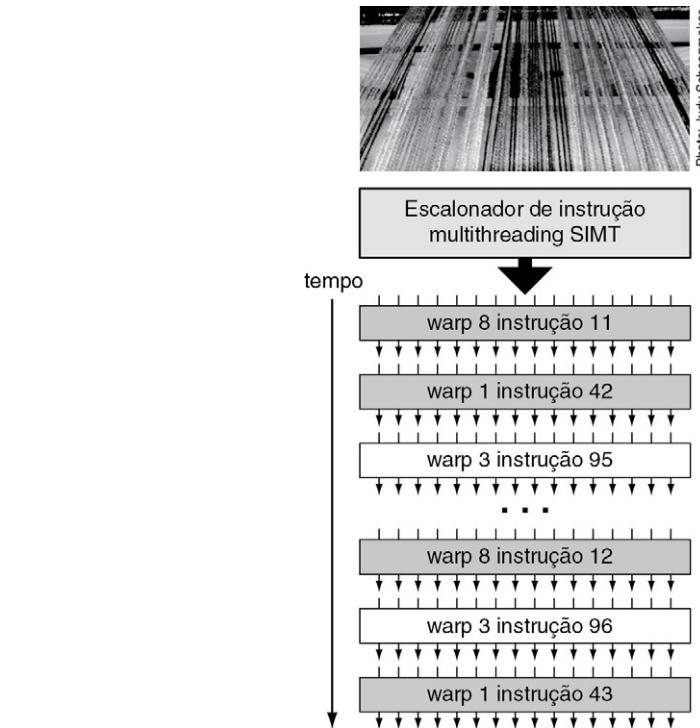


FIGURA A.4.2 Escalonamento de warp multithreaded SIMT. O escalonador seleciona um warp pronto e emite uma instrução sincronamente às threads paralelas compondo o warp. Como os warps são independentes, o escalonador pode selecionar um warp diferente a cada vez.

Esse multiprocessador SIMT de exemplo controla um pool de 16 warps, um total de 512 threads. Threads paralelas individuais compondo um warp são do mesmo tipo e começam juntas no mesmo endereço de programa, mas de outras maneiras são livres para se desviar e executar independentemente. No momento da emissão de cada instrução, a unidade de instrução multithreaded da arquitetura SIMT seleciona um warp que está pronto para executar sua próxima instrução, depois emite essa instrução às threads ativas do warp. Uma instrução SIMT é enviada por broadcast de forma síncrona às threads paralelas ativas de um warp; threads individuais podem ser inativas devido ao desvio ou previsão independente. Nesse multiprocessador, cada core do processador escalar SP executa uma instrução por quatro threads individuais de um warp usando quatro clocks, refletindo a razão 4:1 das threads de warp aos cores.

A arquitetura de processador SIMT é semelhante ao projeto Single-Instruction Multiple Data (SIMD), que aplica uma instrução a múltiplas pistas de dados, mas difere porque SIMD aplica uma instrução a múltiplas threads independentes em paralelo, e não apenas a múltiplas pistas de dados. Uma instrução para um processador SIMD controla um vetor de múltiplas pistas de dados juntas, enquanto uma instrução para um processador SIMT controla uma thread individual, e a unidade de instrução SIMT emite uma instrução a um warp de threads paralelas independentes por eficiência. O processador SIMT encontra o paralelismo em nível de dados entre as threads em tempo de execução, semelhante ao modo como um processador superescalar encontra o paralelismo em nível de instrução entre as instruções em tempo de execução.

Um processador SIMT observa eficiência e desempenho plenos quando todas as threads de um warp tomam o mesmo caminho de execução. Se as threads de um warp divergirem por meio de um desvio condicional dependente dos dados, a execução serializa para cada caminho de desvio tomado, e quando todos os caminhos terminam, as threads convergem para o mesmo caminho de execução. Para caminhos de mesmo tamanho, um bloco de código if-else divergente é 50% eficiente. O multiprocessador utiliza uma pilha de sincronização de desvio para gerenciar threads independentes que divergem e convergem.

Diferentes warps são executadas independentemente em velocidade plena, sem levar em conta se estão executando caminhos de código comuns ou desconexos. Como resultado, GPUs SIMT são muito mais eficientes e flexíveis no código de desvio do que as GPUs anteriores, pois seus warps são muito mais estreitos que a largura SIMD das GPUs anteriores.

Em comparação com as arquiteturas de vetor SIMD, SIMT permite que os programadores escrevam código paralelo em nível de thread para threads individuais independentes, bem como código paralelo de dados para muitas threads coordenadas. Para a exatidão do programa, o programador pode basicamente ignorar os atributos de execução SIMT dos warps; porém, melhorias de desempenho substanciais podem ser observadas cuidado-se para que o código raramente exija que as threads em um warp divirjam. Na prática, isso é semelhante ao papel das linhas de cache nos códigos tradicionais: o tamanho da linha de cache pode ser seguramente ignorado quando se projeta por exatidão, mas deve ser considerado na estrutura de código quando se projeta por desempenho de pico.

Execução e divergência de warp SIMT

A técnica SIMT de escalar warps independentes é mais flexível que o escalonamento de arquiteturas GPU anteriores. Um warp compreende threads paralelos do mesmo tipo: vértice, geometria, pixel ou cálculo. A unidade básica de processamento de sombreamento do fragmento de pixel é o quad de pixel 2 por 2, implementado como quatro threads de sombreamento de pixel. O controlador do multiprocessador empacota os quads de pixel em um warp. Um bloco de threads compreende um ou mais warps. O projeto SIMT compartilha a unidade de busca e emissão de instrução eficientemente por threads paralelos de um warp, mas requer um warp completo de threads ativas para obter eficiência de desempenho completa.

Esse multiprocessador unificado escalona e executa múltiplos tipos de warp simultaneamente, permitindo que execute warps de vértice e pixel simultaneamente. Seu escalonador de warp opera em menos do que a taxa de clock do processador, pois existem quatro pistas de thread por core de processador. Durante cada ciclo de escalonamento, ele seleciona um warp para executar uma instrução de warp SIMT, como mostra a Figura A.4.2. Uma instrução de warp emitida executa como quatro conjuntos de oito threads por quatro ciclos do processador de vazão. O pipeline do processador utiliza vários clocks de latência para concluir cada instrução. Se o número de warps ativos vezes os clocks por warp ultrapassar a latência do pipeline, o programador pode ignorar a latência do pipeline. Para esse multiprocessador, um schedule round-robin de oito warps tem um período de 32 ciclos entre instruções sucessivas para o mesmo warp. Se o programa puder manter 256 threads ativos por multiprocessador, latências de instrução de até 32 ciclos podem ser escondidas de uma thread sequencial individual. Porém, com poucos warps ativos, a profundidade do pipeline do processador torna-se visível e pode causar stall dos processadores.

Um problema de projeto desafiador é implementar o escalonamento de warp com overhead zero para uma mistura dinâmica de diferentes programas de warp e tipos de programa. O escalonador de instruções precisa selecionar um warp a cada quatro clocks para emitir uma instrução por clock por thread, equivalente a um IPC de 1,0 por core de processador. Como os warps são independentes, as únicas dependências estão entre instruções sequenciais do mesmo warp. O escalonador usa um scoreboard de dependência de registrador para qualificar warps cujas threads ativas estão prontas para executar uma instrução. Ele prioriza todos esses warps prontos e seleciona o que possui mais alta prioridade para emissão. A priorização precisa considerar tipo de warp, tipo de instrução e o desejo de ser imparcial para todos os warps ativos.

Gerenciando threads e blocos de threads

O controlador do multiprocessador e a unidade de instrução gerenciam threads e blocos de threads. O controlador aceita solicitações de trabalho e dados de entrada, e arbitra o acesso aos recursos compartilhados, incluindo a unidade de textura, caminho de acesso à memória e caminhos de E/S. Para cargas de trabalhos gráficos, ele cria e gerencia três

tipos de threads gráficas simultaneamente: vértice, geometria e pixel. Cada um dos tipos de trabalho gráfico possui caminhos independentes de entrada e saída. Ele acumula e empacota cada um desses tipos de trabalho de entrada em warps SIMT de threads paralelas executando o mesmo programa de thread. Ele aloca um warp livre, aloca registradores para as threads de warp e inicia a execução do warp no multiprocessador. Cada programa declara sua demanda de registrador por thread; o controlador inicia um warp somente quando ele pode alocar o contador de registrador solicitado para as threads do warp. Quando todas as threads do warp terminam, o controlador desempacota os resultados e libera os registradores e recursos do warp.

array de thread cooperativo (CTA) Um conjunto de threads que executa o mesmo programa de thread e pode cooperar para calcular um resultado. Um CTA de GPU implementa um bloco de threads CUDA.

O controlador cria **arrays de thread cooperativos (CTAs)**, que implementam blocos de threads CUDA como um ou mais warps de threads paralelos. Ele cria um CTA quando pode criar todos os warps CTA e alocar todos os recursos CTA. Além dos threads e registradores, um CTA requer a alocação de memória compartilhada e barreiras. O programa declara as capacidades exigidas, e o controlador espera até que possa alocar essas quantidades antes de iniciar o CTA. Depois, ele cria warps CTA na taxa de escalonamento de warp, de modo que um programa CTA começa a executar imediatamente no desempenho total do multiprocessador. O controlador monitora quando todas as threads de um CTA saíram, e libera os recursos compartilhados do CTA e seus recursos de warp.

Instruções de thread

Os processadores de thread SP executam instruções escalares para threads individuais, diferente das arquiteturas de instrução de vetor da GPU anteriores, que executavam instruções de vetor de quatro componentes para cada programa de sombreamento de vértice ou pixel. Os programas de vértice geralmente calculam vetores de posição (x, y, z, w), enquanto os programas de sombreamento de pixel calculam vetores de cor (vermelho, verde, azul, alfa). Porém, os programas de sombreamento estão se tornando maiores e mais escalares, e é cada vez mais difícil ocupar totalmente até mesmo dois componentes de uma arquitetura de vetor de quatro componentes de GPU legada. Com efeito, a arquitetura SIMD paralleliza por 32 threads de pixel independentes, em vez de parallelizar os quatro componentes de vetor dentro de um pixel. Programas CUDA C/C++ têm código predominantemente escalar por thread. GPUs anteriores empregavam empacotamento de vetor (por exemplo, combinar subvetores de trabalho para ganhar eficiência), mas isso complicava o hardware de escalonamento e também o compilador. Instruções escalares são mais simples e amigáveis a compilador. As instruções de textura continuam sendo baseadas em vetor, tomando um vetor de coordenada de origem e retornando um vetor de cor filtrada.

Para dar suporte a múltiplas GPUs com diferentes formatos de microinstrução binária, gráficos de alto nível e compiladores de linguagem de computação geram instruções intermediárias em nível de assembler (por exemplo, instruções de vetor Direct3D ou instruções escalares PTX), que são então otimizadas e traduzidas para microinstruções GPU binárias. A definição do conjunto de instruções PTX (execução paralela de thread) da NVIDIA [2007] oferece uma ISA de destino estável para compiladores, e oferece compatibilidade por várias gerações de GPUs com arquiteturas de microinstrução binária em evolução. O otimizador prontamente expande as instruções de vetor Direct3D para múltiplas microinstruções binárias escalares.. Instruções escalares PTX são traduzidas quase um para um com microinstruções binárias escalares, embora algumas instruções PTX se expandam para múltiplas microinstruções binárias, e múltiplas instruções PTX possam se desdobrar em uma microinstrução binária. Como as instruções intermediárias em nível de assembler utilizam registradores virtuais, o otimizador analisa as dependências de dados e aloca registradores reais. O otimizador elimina o código morto, reúne instruções quando for viável e optimiza pontos de divergência e convergência de desvio SIMD.

Instruction Set Architecture (ISA)

A thread ISA descrita aqui é uma versão simplificada da arquitetura Tesla PTX ISA, um conjunto de instruções escalar baseado em registrador comprendendo funções de ponto

flutuante, inteiro, lógicas, de conversão, especiais, controle de fluxo, acesso à memória e operações de textura. A [Figura A.4.3](#) lista as instruções de thread PTX GPU básicas; veja detalhes na especificação NVIDIA PTX [2007]. O formato da instrução é:

opcode.type d, a, b, c;

onde d é o operando de destino, a, b, c são operandos de origem, e .type é um destes:

Instruções de thread PTX GPU básicas

Grupo	Instrução	Exemplo	Significado	Comentários
Aritméticas	.tipo aritmético = .s32, .u32, .f32, .s64, .u64, .f64			
	add.tipo	add.f32 d, a, b	d = a + b;	
	sub.tipo	sub.f32 d, a, b	d = a - b;	
	mul.tipo	mul.f32 d, a, b	d = a * b;	
	mad.tipo	mad.f32 d, a, b, c	d = a * b + c;	multiplicação-adição
	div.tipo	div.f32 d, a, b	d = a / b;	múltiplas microinstruções
	rem.tipo	rem.u32 d, a, b	d = a % b;	resto de inteiro
	abs.tipo	abs.f32 d, a	d = a ;	
	neg.tipo	neg.f32 d, a	d = 0 - a;	
	min.tipo	min.f32 d, a, b	d = (a < b)? a:b;	flutuante seleciona não NaN
	max.tipo	max.f32 d, a, b	d = (a > b)? a:b;	flutuante seleciona não NaN
	setp.cmp.tipo	setp.lt.f32 p, a, b	p = (a < b);	compara e define predicado
	.cmp numérico = eq, ne, lt, le, gt, ge	unordered cmp= equ, neu, ltu, leu, gtu, geu, num, nan		
	mov.tipo	mov.b32 d, a	d = a;	move
	selp.tipo	selp.f32 d, a, b, p	d = p? a: b;	seleciona com predicado
	cvt.tipod.tipoa	cvt.f32.s32 d, a	d = convert(a);	converte tipoa para tipod
Função especial	special .tipo = .f32 (some .f64)			
	rcp.tipo	rcp.f32 d, a	d = 1/a;	recíproco
	sqrt.tipo	sqrt.f32 d, a	d = sqrt(a);	raiz quadrada
	rsqrt.tipo	rsqrt.f32 d, a	d = 1/sqrt(a);	raiz quadrada recíproca
	sin.tipo	sin.f32 d, a	d = sin(a);	seno
	cos.tipo	cos.f32 d, a	d = cos(a);	coseno
	lg2.tipo	lg2.f32 d, a	d = log(a)/log(2)	logaritmo binário
	ex2.tipo	ex2.f32 d, a	d = 2 ** a;	exponencial binário
Lógicas	logic.tipo = .pred,.b32, .b64			
	and.tipo	and.b32 d, a, b	d = a & b;	
	or.tipo	or.b32 d, a, b	d = a b;	
	xor.tipo	xor.b32 d, a, b	d = a ^ b;	
	not.tipo	not.b32 d, a, b	d = ~a;	complemento de um
	cnot.tipo	cnot.b32 d, a, b	d = (a==0)? 1:0;	not lógico em C
	shl.tipo	shl.b32 d, a, b	d = a << b;	shift à esquerda
	shr.tipo	shr.s32 d, a, b	d = a >> b;	shift à direita
Acesso à memória	memory .space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.tipo	ld.global.b32 d, [a+off]	d = *(a+off);	load do espaço da memória
	st.space.tipo	st.shared.b32 [d+off], a	* (d+off) = a;	store para espaço da memória
	tex.nd.dtyp.btipo	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	pesquisa de textura
	atom.spc.op.tipo	atom.global.add.u32 d,[a], b	atomic { d = *a;	operação atômica de leitura-modificação-escrita
		atom.global.cas.b32 d,[a], b, c	*a = op(*a, b); }	
Fluxo de controle	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .tipo = .b32			
	branch	@p bra target	if (p) goto target;	desvio condicional
	call	call (ret), func, (params)	ret = func(params);	chamada de função
	ret	ret	return;	retorno de chamada de função
	bar.sync	bar.sync d	wait for threads	sincronização de barreira
	exit	exit	exit;	termina execução da thread

FIGURA A.4.3 Instruções básicas de thread da GPU PTX.

Tipo	Especificador .tipo
Bits não tipados 8, 16, 32 e 64 bits	.b8, .b16, .b32, .b64
Inteiro sem sinal 8, 16, 32 e 64 bits	.u8, .u16, .u32, .u64
Inteiro com sinal 8, 16, 32 e 64 bits	.s8, .s16, .s32, .s64
Ponto flutuante 16, 32 e 64 bits	.f16, .f32, .f64

Operandos de origem são valores escalares de 32 bits ou 64 bits nos registradores, um valor imediato ou uma constante; operandos de predicado são valores booleanos de 1 bit. Destinos são registradores, exceto para store na memória. As instruções têm predicados iniciando-as com @p ou @!p, onde p é um registrador de predicado. Instruções de memória e textura transferem escalares ou vetores de dois a quatro componentes, até 128 bits no total. Instruções PTX especificam o comportamento de um thread.

As instruções aritméticas PTX operam sobre tipos de ponto flutuante, inteiro com sinal e inteiro sem sinal de 32 e 64 bits. GPUs recentes admitem ponto flutuante de precisão dupla com 64 bits; ver Seção A.6. Nas GPUs atuais, instruções PTX com inteiros de 64 bits e lógicas são traduzidas para duas ou mais microinstruções binárias, que realizam operações de 32 bits. As instruções de função especial da GPU são limitadas a ponto flutuante de 32 bits. As instruções de fluxo de controle de thread são branch condicional, call e return de função, exit de thread e bar.sync (sincronização de barreira). A instrução de desvio condicional @p bra target usa um registrador de predicado p (ou !p) definido anteriormente por uma instrução setp de comparação e definição de predicado, para determinar se a thread apanha o desvio ou não. Outras instruções também podem ter predicados em um registrador de predicado verdadeiro ou falso.

Instruções de acesso à memória

A instrução tex busca e filtra amostras de textura de arrays de textura 1D, 2D e 3D na memória por meio do subsistema de textura. As buscas de textura geralmente utilizam coordenadas de ponto flutuante interpoladas para endereçar uma textura. Quando uma thread de sombreamento de pixel gráfico calcula sua cor de fragmento de pixel, o processador de operações de rastreio a mistura com a cor do pixel em sua posição de pixel atribuída (x, y) e escreve a cor final na memória.

Para dar suporte às necessidades de cálculo e da linguagem C/C++, a ISA PTX Tesla implementa as instruções load/store da memória. Elas utilizam endereçamento de inteiros por byte, com aritmética de registrador mais endereço de offset para facilitar as otimizações de código convencionais do compilador. As instruções load/store da memória são comuns nos processadores, mas são uma nova capacidade significativa nas GPUs da arquitetura Tesla, pois GPUs anteriores forneciam apenas a textura e os acessos de pixel exigidos pelas APIs gráficas.

Para cálculo, as instruções load/store acessam três espaços de leitura/escrita que implementam os espaços de memória CUDA correspondentes na Seção A.3:

- Memória local para dados temporários endereçáveis por thread (implementada na DRAM externa)
- Memória compartilhada para acesso de baixa latência aos dados compartilhados por threads em cooperação no mesmo bloco CTA/thread (implementada na SRAM no chip)
- Memória global para grandes conjuntos de dados compartilhados por todas as threads de uma aplicação de cálculo (implementada na DRAM externa)

As instruções load/store da memória ld.global, st.global, ld.shared, st.shared, ld.local e st.local acessam os espaços de memória global, compartilhado e local. Programas de cálculo utilizam a instrução de sincronização de barreira rápida bar.sync para sincronizar threads dentro de um bloco CTA/thread que se comunica um com o outro por meio da memória compartilhada e global.

Para melhorar a largura de banda da memória e reduzir o overhead, as instruções load/store locais e globais juntam solicitações de thread paralelas individuais a partir do mesmo warp SIMT em uma única solicitação de bloco de memória quando os endereços caem no mesmo bloco e atendem critérios de alinhamento. A junção de solicitações de memória oferece um aumento de desempenho significativo em relação a solicitações separadas de threads individuais. A grande contagem de threads do multiprocessador, juntamente com o suporte para muitas solicitações de carga pendentes, ajuda a cobrir a latência de carga para uso da memória local e global implementada na DRAM externa.

As GPUs mais recentes da arquitetura Tesla também oferecem eficientes operações de memória atômicas na memória com as instruções atom.op.u32, incluindo operações com inteiros add, min, max, and, or, xor, exchange e cas (compare-and-swap), facilitando reduções paralelas e gerenciamento de estruturas de dados paralelas.

Sincronização de barreira para comunicação de threads

O sincronismo rápido de barreira permite que programas CUDA se comuniquem frequentemente por meio de memória compartilhada e memória global, simplesmente chamando `_syncthreads()`; como parte de cada etapa de comunicação entre threads. A função de sincronização intrínseca gera uma única instrução `bar.sync`. Porém, implementar a sincronização de barreira rápida entre até 512 threads por bloco de threads CUDA é um desafio.

O agrupamento de threads em warps SIMT de 32 threads reduz a dificuldade de sincronização por um fator de 32. As threads esperam em uma barreira no escalonador de threads SIMT de modo que não consumam quaisquer ciclos de processador enquanto esperam. Quando uma thread executar uma instrução `bar.sync`, ela incrementa o contador de chegada de thread da barreira e o escalonador marca a thread como esperando na barreira. Quando todas as threads CTA chegarem, o contador da barreira é igual ao contador terminal esperado, e o escalonador libera todas as threads esperando na barreira e continua executando threads.

Streaming Processor (SP)

O processador streaming (SP) multithreaded é o principal processador de instruções de thread no multiprocessador. Seu arquivo de registradores (RF) oferece 1024 registradores escalares de 32 bits para até 64 threads. Ele executa todas as operações fundamentais de ponto flutuante, incluindo `add.f32`, `mul.f32`, `mad.f32` (floating multiply-add), `min.f32`, `max.f32` e `sept.f32` (floating compare and set predicate). As operações de adição e multiplicação em ponto flutuante são compatíveis com o padrão IEEE 754 para números de PF em precisão simples, incluindo valores not-a-number (NaN) e infinito. O core SP também implementa todas as instruções PTX aritméticas, de comparação, conversão e lógicas com inteiros de 32 e 64 bits na [Figura A.4.3](#).

As operações de ponto flutuante `add` e `mul` empregam o arredondamento-par-mais-próximo do IEEE como modo de arredondamento padrão. A operação de multiplicação-adição de ponto flutuante `mad.f32` realiza uma multiplicação truncada, seguida por uma adição com arredondamento-par-mais-próximo. O SP limpa os operandos des-normalizados da entrada para zero-com-sinal-preservado. Os resultados com underflow da faixa de expoentes da saída de destino são limpos para zero-com-sinal-preservado após o arredondamento.

Special Function Unit (SFU)

Certas instruções de thread podem ser executadas nas SFUs, simultaneamente com outras instruções de thread executando nos SPs. A SFU implementa as instruções de função especial da [Figura A.4.3](#), que calcula aproximações de ponto flutuante de 32 bits para funções transcendentais recíprocas, de raiz quadrada recíproca e de chave. Ela também implementa interpolação de atributo planar em ponto flutuante para sombreamentos de pixel, oferecendo interpolação precisa de atributos como coordenadas de cor, profundidade e textura.

Cada SFU em pipeline gera um resultado de função especial de ponto flutuante de 32 bits por ciclo; as duas SFUs por multiprocessador executam instruções de função especial em um quarto da taxa de instrução simples dos oito SPs. As SFUs também executam a instrução de multiplicação mul.f32 simultaneamente com os oito SPs, aumentando a taxa de cálculo máxima para 50%, para threads com uma mistura de instruções adequada.

Por avaliação funcional, a SFU da arquitetura Tesla emprega interpolação quadrática, com base nas aproximações minimax avançadas, para aproximar as funções recíprocas, raiz quadrada recíproca, $\log_2 x$, 2^x e \sin/\cos . A precisão da função estima intervalos de 22 a 24 bits de mantissa. Veja mais detalhes sobre aritmética de SFU na Seção A.6.

Comparando com outros multiprocessadores

Em comparação com as arquiteturas de vetor SIMD, como SSE x86, o multiprocessador SIMT pode executar threads individuais independentemente, em vez de sempre executá-las juntas em grupos síncronos. O hardware SIMT encontra paralelismo de dados entre threads independentes, enquanto o hardware SIMD requer que o software expresse o paralelismo de dados explicitamente em cada instrução de vetor. Uma máquina SIMT executa um warp de 32 threads de forma síncrona quando as threads tomam o mesmo caminho de execução, embora possa executar cada thread independentemente quando elas divergirem. A vantagem é significativa, pois os programas e as instruções SIMT simplesmente descrevem o comportamento de uma única thread independente, ao em vez de um vetor de dados SIMD de quatro ou mais pistas de dados. Apesar disso, o multiprocessador SIMT possui eficiência tipo SIMD, espalhando a superfície e o custo de uma unidade de instrução pelas 32 threads de um warp e pelos oito cores de processador streaming. SIMT oferece o desempenho do SIMD junto com a produtividade do multithreading, evitando a necessidade de codificar explicitamente os vetores SIMD do código para condições de aresta e divergência parcial.

O multiprocessador SIMT impõe pouco overhead, pois é multithreaded por hardware com sincronização de barreira de hardware. Isso permite que sombreamentos gráficos e threads CUDA expressem um paralelismo bastante fine-grained. Programas gráficos e CUDA utilizam threads para expressar paralelismo de dados fine-grained em um programa por thread, em vez de forçar o programador a expressá-lo como instruções de vetor SIMD. É mais simples e mais produtivo desenvolver código de única thread escalar que o código de vetor, e o multiprocessador SIMT executa o código com eficiência tipo SIMD.

Juntar oito cores de processador streaming em um multiprocessador e depois implementar um número escalável desses multiprocessadores cria um multiprocessador de dois níveis composto de multiprocessadores. O modelo de programação CUDA explora a hierarquia de dois níveis oferecendo threads individuais para cálculos paralelos fine-grained, e oferecendo grades de blocos de thread para operações paralelas coarse-grained. O mesmo programa de thread pode oferecer operações fine-grained e coarse-grained. Ao contrário, as CPUs com instruções de vetor SIMD precisam usar dois modelos de programação diferentes para oferecer operações fine-grained e coarse-grained: threads paralelas coarse-grained nos diferentes cores, e instruções de vetor SIMD para o paralelismo de dados fine-grained.

Conclusão sobre multiprocessador multithreaded

O multiprocessador da GPU de exemplo baseado na arquitetura Tesla é altamente multithreaded, executando um total de até 512 threads peso leve simultaneamente para dar suporte a sombreamentos de pixel fine-grained e threads CUDA. Ele utiliza uma variação da arquitetura SIMD e multithreading, chamada SIMT (Single-Instruction Multiple-Thread) para enviar uma instrução por broadcast, de forma eficiente, para um warp de 32 threads paralelas, enquanto permite que cada thread se desvie e seja executada independentemente. Cada thread executa seu fluxo de instruções em um dos oito cores do processador streaming (SP), que são multithreaded para até 64 threads.

A ISA PTX é uma ISA escalar de load/store baseada em registrador, que descreve a execução de uma única thread. Como as instruções PTX são otimizadas e traduzidas para microinstruções binárias para uma GPU específica, as instruções de hardware podem evoluir rapidamente sem atrapalhar compiladores e ferramentas de software que geram instruções PTX.

A.5

Sistema de memória paralela

Fora a própria GPU, o subsistema de memória é o determinante mais importante do desempenho de um sistema gráfico. As cargas de trabalhos gráficos exigem taxas de transferência muito altas de e para a memória. Operações de escrita e blend (ler-modifica-escrever) de pixel, leituras e escritas de buffer em profundidade, leituras de mapa de textura, além de leituras de dados de vértice e atributo de objeto, compreendem a maior parte do tráfego da memória.

As GPUs modernas são altamente paralelas, como mostramos na [Figura A.2.5](#). Por exemplo, o GeForce 8800 pode processar 32 pixels por clock, a 600MHz. Cada pixel normalmente requer uma leitura e escrita de cor e uma leitura e escrita de profundidade de um pixel de 4 bytes. Normalmente, uma média de dois ou três texels de quatro bytes cada são lidos para gerar a cor do pixel. Assim, para um caso típico, existe uma demanda de 28 bytes vezes 32 pixels = 896 bytes por clock. Nitidamente, a demanda de largura de banda no sistema de memória é enorme.

Para fornecer esses requisitos, os sistemas de memória da GPU têm as seguintes características:

- Eles são amplos, significando que existe um grande número de pinos para transmitir dados entre a GPU e seus dispositivos de memória, e o próprio array de memória comprehende muitos chips DRAM para fornecer a largura total do barramento de dados.
- Eles são rápidos, significando que técnicas de sinalização agressivas são usadas para maximizar a taxa de dados (bits/segundo) por pino.
- As GPUs buscam usar cada ciclo disponível para transferir dados de e para o array de memória. Para conseguir isso, as GPUs especificamente não visam minimizar a latência ao sistema de memória. Alta vazão (eficiência de utilização) e latência curta estão fundamentalmente em conflito.
- Técnicas de compactação são utilizadas, tanto com perdas, das quais o programador precisa estar ciente, quanto sem perdas, que é invisível à aplicação e oportunista.
- Caches e estruturas de junção de trabalho são usadas para reduzir a quantidade de tráfego fora do chip necessário e garantir que os ciclos gastos movendo-se dados são usados o mais totalmente possível.

Considerações sobre DRAM

As GPUs precisam levar em consideração as características exclusivas da DRAM. Chips de DRAM são arrumados internamente como múltiplos bancos (normalmente, de quatro a oito), em que cada banco inclui um número de linhas na potência de 2 (normalmente, por volta de 16.384), e cada linha contém um número de bits na potência de dois (normalmente, 8192). As DRAMs impõem uma série de requisitos de temporização em seu processador de controle. Por exemplo, dezenas de ciclos são exigidos para ativar uma linha, mas, uma vez ativados, os bits dentro dessa linha são acessíveis aleatoriamente com um novo endereço de coluna a cada quatro clocks. DRAMs síncronas Double-Data Rate (DDR) transferem dados nas arestas de subida e descida do clock de interface (veja Capítulo 5). Assim, uma

DRAM DDR com clock de 1GHz transfere dados a 2 gigabits por segundo por pino de dados. DRAMs DDR gráficas normalmente possuem 32 pinos de dados bidirecionais, de modo que oito bytes podem ser lidos ou escritos a partir da DRAM por clock.

As GPUs internamente possuem um grande número de geradores de tráfego de memória. Diferentes estágios do pipeline gráfico lógico possuem cada um seus próprios streams de solicitação: busca de atributo de comando e vértice, busca e load/store de textura de sombreamento, e leitura-escrita de profundidade e cor de pixel. Em cada estágio lógico, normalmente existem múltiplas unidades independentes para oferecer a vazão paralela. Estas são solicitadores de memória independentes. Quando vistas no sistema de memória, existe um número enorme de solicitações não correlacionadas durante a execução. Essa é uma divergência natural do padrão de referência preferido pelas DRAMs. Uma solução é que o controlador de memória da GPU mantenha heaps separadas de tráfego voltado para diferentes bancos de DRAM, e esperem até que um tráfego suficiente para determinada linha da DRAM esteja pendente antes de ativar essa linha e transferir todo o tráfego ao mesmo tempo. Observe que acumular solicitações pendentes, embora seja bom para a localidade de linha de DRAM e, portanto, para o uso eficiente do barramento de dados, leva a uma latência média mais longa, conforme visto pelos solicitantes cujas solicitações gastam tempo esperando por outras. O projeto precisa cuidar para que nenhuma solicitação em particular espere muito tempo, ou então algumas unidades de processamento podem “morrer de fome” esperando por dados e por fim fazer com que os processadores vizinhos se tornem ociosos.

Os subsistemas de memória da GPU são arrumados como múltiplas *partições de memória*, cada um compreendendo um controlador de memória totalmente independente e um ou dois dispositivos de DRAM que são totalmente e exclusivamente possuídos por essa partição. Para conseguir o melhor balanceamento de carga e, portanto, aproximar o desempenho teórico de n partições, os endereços são intercalados detalhadamente por todas as partições de memória. O caminho de intercalação da partição em geral é um bloco de algumas centenas de bytes. O número de partições de memória é projetado para balancear o número de processadores e outros solicitadores de memória.

Caches

As cargas de trabalho da GPU normalmente possuem conjuntos de trabalho muito grandes – na ordem de centenas de megabytes para gerar um único frame gráfico. Diferente das CPUs, não é prático construir caches em chips grandes o suficiente para manter qualquer coisa próxima do conjunto de trabalho inteiro de uma aplicação gráfica. Enquanto as CPUs podem assumir taxas de acerto de cache muito altas (99,9% ou mais), as GPUs experimentam taxas de acerto mais próximas de 90% e, portanto, precisam lidar com muitas falhas durante a execução. Embora uma CPU possa ser razoavelmente projetada para interromper enquanto espera por uma falha de cache rara, uma GPU precisa prosseguir com falhas e acertos misturados. Chamamos isso de *arquitetura de cache streaming*.

Caches da GPU precisam oferecer largura de banda muito alta aos seus clientes. Considere o caso de uma cache de textura. Uma unidade de textura comum pode avaliar duas interpolações bilineares para cada um dos quatro pixels por ciclo de clock, e uma GPU pode ter muitas dessas unidades de textura, todas operando independentemente. Cada interpolação bilinear requer quatro texels separados, e cada texel poderia ser um valor de 64 bits. É mais comum ter quatro componentes de 16 bits. Assim, a largura de banda total é $2 \times 4 \times 4 \times 64 = 2048$ bits por clock. Cada texel de 64 bits separado é endereçado independentemente, de modo que a cache precisa tratar de 32 endereços exclusivos por clock. Isso naturalmente favorece um arranjo multibanco e/ou multiporta de arrays SRAM.

MMU

GPUs modernas são capazes de traduzir endereços virtuais para endereços físicos. No GeForce 8800, todas as unidades de processamento geram endereços de memória em um

espaço de endereço virtual de 40 bits. Para cálculo, as instruções de thread de load e store utilizam endereços de byte com 32 bits, que são estendidos para um endereço virtual de 40 bits acrescentando um offset de 40 bits. Uma unidade de gerenciamento de memória realiza tradução de endereço de virtual para físico; o hardware lê as tabelas de página da memória local para responder a falhas em favor de uma hierarquia de buffers lookaside de tradução espalhados entre os processadores e mecanismos de renderização. Além dos bits da página física, as entradas da tabela de página da GPU especificam o algoritmo de compactação para cada página. Os tamanhos de página variam de 4 a 128 kilobytes.

Espaços de memória

Conforme apresentado na Seção A.3, CUDA expõe diferentes espaços de memória para permitir que o programador armazene valores de dados de uma forma que seja ideal para o desempenho. Para a discussão a seguir, assumimos GPUs da arquitetura Tesla da NVIDIA.

Memória global

A memória global é armazenada na DRAM externa; ela não é local a qualquer multiprocessador de streaming (SM) físico isolado, pois visa a comunicação entre diferentes CTAs (blocos de threads) em diferentes grades. Na verdade, os muitos CTAs que referenciam um local na memória global podem não estar executando na GPU ao mesmo tempo; por projeto, em CUDA, um programador não sabe a ordem relativa em que os CTAs são executados. Como o espaço de endereço é distribuído uniformemente entre todas as partições de memória, é preciso haver um caminho de leitura/escrita de qualquer multiprocessador streaming para qualquer partição da DRAM.

O acesso à memória global por diferentes threads (e diferentes processadores) não tem garantias de ter consistência sequencial. Os programas com threads veem um modelo de ordenação de memória relaxado. Dentro de uma thread, a ordem das leituras e escritas na memória para o mesmo endereço é preservada, mas a ordem dos acessos a diferentes endereços pode não ser preservada. As leituras e escritas na memória solicitadas por diferentes threads são desordenadas. Dentro de um CTA, a instrução de sincronização de barreira `bar.sync` pode ser usada para obter ordenação de memória estrita entre as threads do CTA. A instrução de thread `membar` oferece uma operação de barreira/cerca de memória que valida os acessos anteriores à memória e os torna visíveis a outras threads antes de prosseguir. As threads também podem usar as operações atômicas da memória descritas na [Seção A.4](#) para coordenar o trabalho na memória que elas compartilham.

Memória compartilhada

A memória compartilhada por CTA só é visível às threads que pertencem a esse CTA, e a memória compartilhada só ocupa armazenamento a partir do momento em que um CTA é criado até o momento em que ele termina. A memória compartilhada, portanto, pode residir no chip. Essa técnica tem muitos benefícios. Primeiro, o tráfego da memória compartilhada não precisa competir com a largura de banda limitada fora do chip, necessária para referências à memória global. Segundo, é prático criar estruturas de memória com largura de banda muito alta no chip para dar suporte às demandas de leitura/escrita de cada multiprocessador streaming. De fato, a memória compartilhada é bastante acoplada ao multiprocessador streaming.

Cada multiprocessador streaming contém oito processadores de thread físicos. Durante um ciclo de memória compartilhada, cada processador de thread pode processar dois threads de instruções, de modo que 16 threads de solicitações de memória compartilhada precisam ser tratados em cada clock. Como cada thread pode gerar seus próprios endereços, e os endereços normalmente são exclusivos, a memória compartilhada é montada usando-se 16 bancos SRAM endereçáveis independentemente. Para os padrões de acesso comuns, 16 bancos são suficientes para manter a vazão, mas casos patológicos são possíveis; por exemplo, todas as 16 threads poderiam acessar um endereço diferente

em um banco de SRAM. Deverá ser possível rotear uma solicitação a partir de qualquer pista de thread para qualquer banco de SRAM, de modo que é exigida uma rede de interconexão de 16 por 16.

Memória local

A memória local por thread é a memória particular visível apenas a uma única thread. A memória local é arquitetonicamente maior que o arquivo de registrador da thread, e um programa pode calcular endereços na memória local. Para dar suporte a grandes alocações de memória local (lembre-se de que a alocação total é a alocação por thread vezes o número de threads ativas), a memória local é alocada na DRAM externa.

Embora a memória global e a local por thread residam fora do chip, elas são bem adequadas a serem mantidas em cache no chip.

Memória constante

A memória constante é somente de leitura para um programa rodando no SM (ela pode ser escrita por meio de comandos à GPU). Ela é armazenada na DRAM externa e mantida em cache no SM. Como normalmente a maioria ou todas as threads em um warp SIMT leem do mesmo endereço na memória constante, uma única pesquisa de endereço por clock é suficiente. A cache constante é projetada para enviar valores escalares por broadcast às threads em cada warp.

Memória de textura

A memória de textura mantém grandes arrays de dados somente de leitura. As texturas para computação têm os mesmos atributos e capacidades das texturas usadas com gráficos 3D. Embora as texturas sejam normalmente imagens bidimensionais (arrays 2D de valores de pixel), texturas 1D (lineares) e 3D (volume) também estão disponíveis.

Um programa de cálculo referencia uma textura usando uma instrução `tex`. Os operandos incluem um identificador para nomear a textura, e 1, 2 ou 3 coordenadas, com base na dimensionalidade da textura. As coordenadas de ponto flutuante incluem uma parte fracionária que especifica um local de amostra normalmente entre os locais de texel. Coordenadas não inteiras invocam uma interpolação ponderada bilinear dos quatro valores mais próximos (para uma textura 2D) antes que o resultado seja retornado ao programa.

Buscas de textura são mantidas em cache em uma hierarquia de cache streaming projetada para otimizar a vazão das buscas de textura de milhares de threads simultâneas. Alguns programas usam buscas de textura como um modo de manter a memória global em cache.

Superfícies

Superfície é um termo genérico para um array unidimensional, bidimensional ou tridimensional de valores de pixel e um formato associado. Diversos formatos são definidos; por exemplo, um pixel pode ser definido como quatro componentes inteiros RGBA de 8 bits, ou quatro componentes de ponto flutuante de 16 bits. Um kernel de programa não precisa conhecer o tipo da superfície. Uma instrução `tex` converte seus valores de resultado como ponto flutuante, dependendo do formato da superfície.

Acesso de load/store

As instruções load/store com endereçamento de byte inteiro permitem a escrita e compilação de programas em linguagens convencionais, como C e C++. Programas CUDA utilizam instruções load/store para acessar a memória.

Para melhorar a largura de banda da memória e reduzir o overhead, as instruções load/store globais juntam solicitações de thread paralelas individuais a partir do mesmo warp em uma única solicitação de bloco de memória quando os endereços caem no mesmo

bloco e atendem aos critérios de alinhamento. Juntar solicitações de memória pequenas em solicitações de bloco maiores oferece um aumento de desempenho significativo sobre solicitações separadas. A grande quantidade de threads, junto com o suporte para muitas solicitações de load pendentes, ajuda a cobrir a latência de load-para-uso para a memória local e global implementada na DRAM externa.

ROP

Como podemos ver na [Figura A.2.5](#), GPUs da arquitetura Tesla NVIDIA compreendem um array de processador streaming (SPA), que realiza todos os cálculos programáveis da GPU, e um sistema de memória escalável, que compreende controle de DRAM externa e Raster Operation Processors (ROPs) de função fixa, que realizam operações de buffer de frame de cor e profundidade diretamente na memória. Cada unidade ROP é emparelhada com uma partição de memória específica. Partições ROP são alimentadas a partir dos SMs por meio de uma rede de interconexão. Cada ROP é responsável por testes e atualizações de profundidade e estêncil, além de mistura de cores. Os controladores de ROP e memória cooperam para implementar compactação de cor e profundidade sem perda (até 8:1) para reduzir a largura de banda externa. Unidades ROP também realizam operações atômicas na memória.

A.6

Aritmética de ponto flutuante

As GPUs hoje realizam a maioria das operações aritméticas nos cores de processador programáveis usando operações de ponto flutuante de 32 bits com precisão simples compatíveis com IEEE 754 (veja Capítulo 3). A aritmética de ponto flutuante das primeiras GPUs foi sucedida por ponto flutuante de 16 bits, 24 bits e 32 bits, depois ponto flutuante de 32 bits compatível com IEEE 754. Alguma lógica de função fixa dentro de uma GPU, como hardware de filtragem de textura, continua a usar os formatos numéricos proprietários. GPUs recentes também oferecem instruções de ponto flutuante de 64 bits com precisão dupla compatível com IEEE 754.

Formatos aceitos

O padrão IEEE 754 para aritmética de ponto flutuante [2008] especifica formatos básicos e de armazenamento. As GPUs usam dois dos formatos básicos para computação, ponto flutuante binário de 32 e 64 bits, normalmente chamados de precisão simples e precisão dupla. O padrão também especifica um formato de ponto flutuante de armazenamento binário de 16 bits, **meia precisão**. GPUs e a linguagem de sombreamento Cg empregam o formato de dados estreito de 16 bits para armazenamento e movimentação de dados eficiente, embora mantendo alta faixa dinâmica. GPUs realizam muitos cálculos de filtragem de textura e mistura de pixel em meia precisão dentro da unidade de filtragem de textura e a unidade de operações de rastreio. O formato de arquivo de imagem de alta faixa dinâmica OpenEXR, desenvolvido pela Industrial Light and Magic [2003], usa o formato de metade idêntico para valores de componente de cor em aplicações de imagens de computador e desenho animado.

meia precisão Um formato de ponto flutuante binário de 16 bits, com 1 bit de sinal, expoente de 5 bits, fração de 10 bits e um bit de inteiro implícito.

Aritmética básica

As operações comuns de ponto flutuante com precisão simples em cores programáveis da GPU incluem adição, multiplicação, **multiplicação-adição**, mínimo, máximo, comparação, definição de predicado e conversões entre números inteiros e de ponto flutuante. As instruções de ponto flutuante normalmente oferecem modificadores de operando de origem para negação e valor absoluto.

multiplicação-adição (MAD) Uma instrução única em ponto flutuante que realiza uma operação composta: multiplicação seguida por adição.

As operações de adição e multiplicação em ponto flutuante da maioria das GPUs atualmente são compatíveis com o padrão IEEE 754 para números de PF de precisão simples, incluindo not-a-number (NaN) e valores infinitos. As operações de adição e multiplicação de PF utilizam o arredondamento-para-par-mais-próximo como o modo de arredondamento default. Para aumentar a vazão da instrução em ponto flutuante, as GPUs normalmente utilizam uma instrução de multiplicação-adição composta (mad). A operação de multiplicação-adição de ponto flutuante realiza multiplicação de PF com truncamento, seguida por adição de PF com arredondamento-para-para-mais-próximo. Ela oferece duas operações de ponto flutuante em um ciclo de emissão, sem exigir que o escalonador de instrução despache duas instruções separadas, mas o cálculo não é fundido e trunca o produto antes da adição. Isso a torna diferente da instrução de multiplicação-adição fundida, discutida no Capítulo 3, e mais adiante nesta seção. GPUs normalmente limpam operandos de origem desnormalizados para zero preservado por sinal, e eles limpam resultados que passam por underflow da faixa de expoente de saída de destino para zero preservado por sinal após o arredondamento.

Aritmética especializada

As GPUs oferecem hardware para acelerar o cálculo de função especial, interpolação de atributo e filtragem de textura. Instruções de função especial incluem cosseno, seno, exponencial binário, logaritmo binário, recíproco e raiz quadrada recíproca. Instruções de interpolação de atributo oferecem geração eficiente de atributos de pixel, derivados da avaliação da equação do plano. A **unidade de função especial (SFU)** introduzida na Seção A.4 calcula funções especiais e interpola atributos planares [Oberman e Siu, 2005].

Existem vários métodos para avaliar funções especiais no hardware. Mostrou-se que a interpolação quadrática baseada em Enhanced Minimax Approximations é um método muito eficiente para aproximar funções no hardware, incluindo recíproco, raiz quadrada recíproca, $\log_2 x$, 2^x , seno e cosseno.

Podemos resumir o método de interpolação quadrática SFU. Para um operando de entrada binário X com significando de n bits, o significando é dividido em duas partes: X_u é a parte superior, contendo m bits, e X_l é a parte inferior, contendo $n-m$ bits. Os m bits superiores X_u são usados para consultar um conjunto de três tabelas de pesquisa para retornar três coeficientes de palavra finita C_0 , C_1 e C_2 . Cada função a ser aproximada requer um conjunto exclusivo de tabelas. Esses coeficientes são usados para aproximar determinada função $f(X)$ no intervalo $X_u \leq X < X_u + 2^{-m}$ avaliando a expressão:

$$f(X) = C_0 + C_1 X_l + C_2 X_l^2$$

A precisão de cada função estima os intervalos de 22 a 24 bits significativos. As estatísticas de função de exemplo aparecem na Figura A.6.1.

Função	Intervalo de entrada	Precisão (bits bons)	Erro ULP*	% arredondada exatamente	Monotônico
$1/x$	[1, 2)	24,02	0,98	87	Sim
$1/\sqrt{x}$	[1, 4)	23,40	1,52	78	Sim
2^x	[0, 1)	22,51	1,41	74	Sim
$\log_2 x$	[1, 2)	22,57	N/A**	N/A	Sim
\sin/\cos	[0, $\pi/2$)	22,47	N/A	N/A	Sim

*ULP ou UUL: unidade no último local. **N/A: não se aplica.

FIGURA A.6.1 Estatísticas de aproximação de função especial. Para a unidade de função especial (SFU) do NVIDIA GeForce 8800.

O padrão IEEE 754 especifica requisitos de arredondamento exato para divisão e raiz quadrada; contudo, para muitas aplicações de GPU, a compatibilidade exata não é exigida. Em vez disso, para essas aplicações, a vazão de cálculo mais alta é mais importante do que a precisão até o último bit. Para as funções especiais da SFU, a biblioteca de matemática CUDA oferece uma função de precisão completa e uma função rápida com a precisão da instrução SFU.

Outra operação aritmética especializada em uma GPU é a interpolação de atributo. Os principais *atributos* normalmente são especificados para vértices de primitivos que compõem uma cena a ser renderizada. Exemplos de atributos são coordenadas de cor, profundidade e textura. Esses atributos precisam ser interpolados no espaço de tela (x, y) conforme a necessidade, para determinar os valores dos atributos em cada local de pixel. O valor de determinado atributo U em um plano (x, y) pode ser expresso usando-se equações de plano na forma:

$$U(x, y) = A_u x + B_u y + C_u$$

onde A, B e C são parâmetros de interpolação associadas a cada atributo U . Os parâmetros de interpolação A, B e C são todos representados como números de ponto flutuante de precisão simples.

Dada a necessidade para um avaliador de função e um interpolador de atributo em um processador de sombreamento de pixel, uma única SFU que realiza as duas funções por eficiência poderá ser projetada. As duas funções usam uma operação de soma de produtos para interpolar resultados, e o número de termos a ser resumido nas duas funções é muito semelhante.

Operações de textura

Mapeamento e filtragem de textura é outro conjunto principal de operações aritméticas de ponto flutuante especializadas em uma GPU. As operações usadas para mapeamento de textura incluem:

1. Receber endereço de textura (s, t) para o pixel de tela atual (x, y), onde s e t são números de ponto flutuante de precisão simples.
2. Calcular o nível de detalhe para identificar o **nível de mapa MIP** com textura correta.
3. Calcular a fração de interpolação trilinear.
4. Escalar endereço de textura (s, t) para o nível de mapa de MIP selecionado.
5. Acessar memória e recuperar os texels desejados (elementos de textura).
6. Realizar operação de filtragem sobre os texels.

mapa de MIP Uma frase em Latin *multum in parvo*, ou muito em um espaço pequeno. Um mapa de MIP contém imagens pré-calculadas de diferentes resoluções, usadas para aumentar a velocidade de renderização e reduzir artefatos.

O mapeamento de textura requer uma quantidade significativa de cálculo de ponto flutuante para a operação em velocidade plena, grande parte feita em meia precisão com 16 bits. Como um exemplo, o GeForce 8800 Ultra emite cerca de 500 GFLOPS de cálculo de ponto flutuante em formato próprio para instruções de mapeamento de textura, além de suas instruções convencionais IEEE de ponto flutuante com precisão simples. Para obter mais detalhes sobre mapeamento e filtragem de textura, consulte Foley e van Dam [1995].

Desempenho

O hardware aritmético de adição e multiplicação em ponto flutuante utiliza pipelines em sua totalidade, e a latência é otimizada para balancear atraso e área. Em pipeline, a vazão das funções especiais é menor que as operações de adição e multiplicação em ponto flutuante. A vazão em um quarto de velocidade para as funções especiais é o desempenho típico nas GPUs modernas, com uma SFU compartilhada por quatro cores do SP. Ao contrário, as CPUs normalmente têm uma vazão significativamente menor para funções semelhantes,

como divisão e raiz quadrada, embora com resultados mais exatos. O hardware de interpolação de atributo normalmente utiliza pipelines em sua totalidade, para permitir sombreamentos de pixel em velocidade plena.

Dupla precisão

GPUs mais recentes, como a Tesla T10P, também admitem operações de precisão dupla IEEE 754 com 64 bits no hardware. As operações aritméticas de ponto flutuante padrão em precisão dupla incluem adição, multiplicação e conversões entre diferentes formatos de ponto flutuante e inteiro. O padrão de ponto flutuante IEEE 754 de 2008 inclui especificação para a operação fundida de multiplicação-adição reunida (FMA), conforme discutimos no Capítulo 3. A operação FMA realiza uma multiplicação de ponto flutuante seguida por uma adição, com um único arredondamento. As operações de multiplicação e adição fundidas retêm precisão total nos cálculos intermediários. Esse comportamento permite cálculos de ponto flutuante mais precisos, envolvendo o acúmulo de produtos, incluindo produtos de ponto, multiplicação de matriz e avaliação polinomial. A instrução FMA também permite implementações de software eficientes de divisão e raiz quadrada arredondadas exatamente, evitando a necessidade de uma unidade de divisão ou raiz quadrada no hardware.

Uma unidade FMA de precisão dupla no hardware implementa adição, multiplicação e conversões com 64 bits, além da própria operação FMA. A arquitetura de uma unidade FMA de precisão dupla permite o suporte para número desnormalizado em velocidade plena nas entradas e saídas. A [Figura A.6.2](#) mostra um diagrama em blocos de uma unidade FMA.

Como podemos ver na [Figura A.6.2](#), os significados de A e B são multiplicados para formar um produto de 106 bits, com os resultados mantidos na forma de carry-save (salvar vai-um). Em paralelo, o somando de 53 bits C é condicionalmente invertido e alinhado ao produto de 106 bits. Os resultados de soma e vai-um do produto de 106 bits são somados

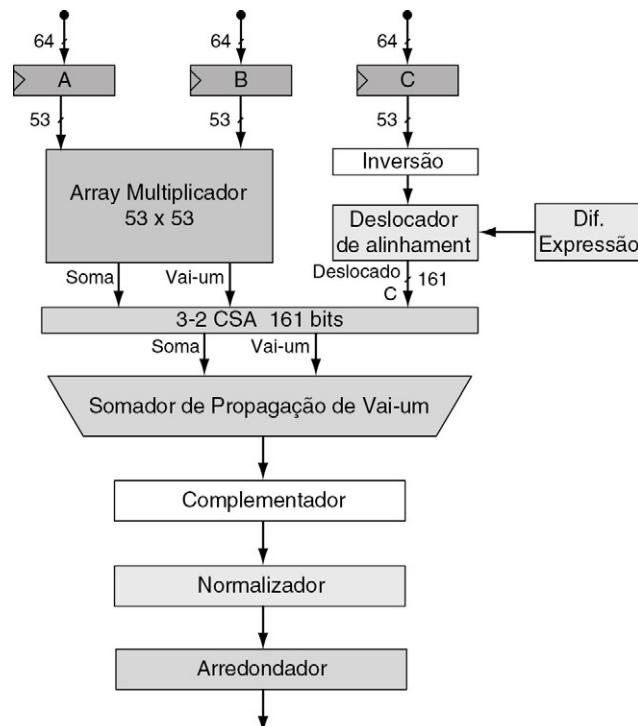


FIGURA A.6.2 Unidade fundida de multiplicação-adição (FMA) com precisão dupla. Hardware para implementar $A \times B + C$ em ponto flutuante para precisão dupla.

com o somando alinhado por um somador carry-save (CSA) com 161 bits de largura. A saída do carry-save é então somada em um somador carry-propagate (propagar vai-um) para produzir um resultado não arredondando na forma não redundante, de complemento de dois. O resultado é recomplementado condicionalmente, de modo a retornar um resultado na forma de magnitude com sinal. O resultado complementado é normalizado, e depois é arredondado para caber dentro do formato de destino.

A.7

Vida real: o NVIDIA GeForce 8800

O NVIDIA GeForce 8800 GPU, introduzido em novembro de 2006, é um projeto de processador unificado de vértice e pixel, que também admite aplicações de cálculo paralelo escritas em C usando o modelo de programação paralela CUDA. Essa é a primeira implementação da arquitetura unificada de gráficos e cálculo descrita na [Seção A.4](#) e em Lindholm, Nickolls, Oberman e Montrym [2008]. Uma família de GPUs da arquitetura Tesla enfoca as diferentes necessidades dos laptops, desktops, estações de trabalho e servidores.

Array de processador streaming (SPA)

A GPU GeForce 8800 mostrada na [Figura A.7.1](#) contém 128 cores de processador streaming (SP) organizados como 16 multiprocessadores streaming (SMs). Dois SMs compartilham uma unidade de textura em cada cluster textura/processador (TPC). Um array de oito TPCs compõe o array de processador streaming (SPA), que executa todos os programas de sombreamento gráfico e programas de cálculo.

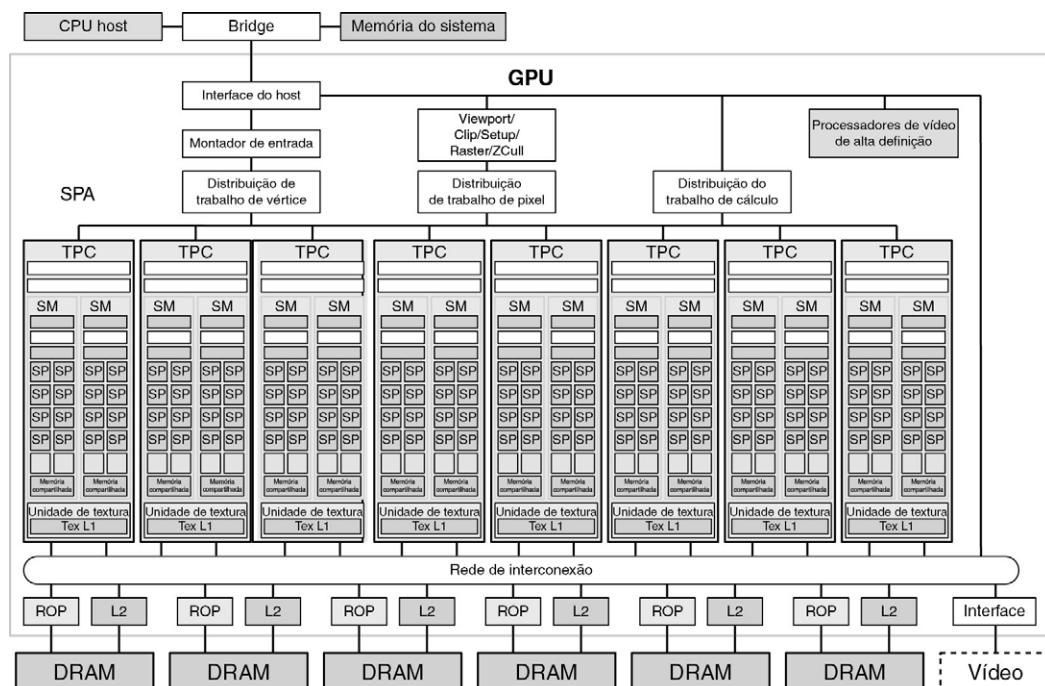


FIGURA A.7.1 Arquitetura unificada de gráficos e cálculo da GPU NVIDIA Tesla. Esse GeForce 8800 tem 128 cores de processador streaming (SP) em 16 multiprocessadores streaming (SM), arrumados em oito clusters de textura/processador (TPC). Os processadores se conectam a seis partições DRAM com 64 bits de largura por meio de uma rede de interconexão. Outras GPUs implementando a arquitetura Tesla variam o número de cores SP, SMs, partições de DRAM e outras unidades.

A unidade de interface host se comunica com a CPU host por meio do barramento PCI-Express, verifica a consistência do comando e realiza a mudança de contexto. O montador de entrada coleta primitivos geométricos (ponto, linhas, triângulos). Os blocos de distribuição de trabalho despacham vértices, pixels e arrays de threads de cálculo para os TPCs no SPA. Os TPCs executam programas de sombreamento de vértice e geometria e programas de cálculo. Os dados geométricos de saída são enviados ao bloco viewport/clip/setup/raster/zcull para serem rasterizados em fragmentos de pixel, que são então redistribuídos de volta ao SPA para a execução de programas de sombreamento de pixel. Os pixels sombreados são enviados pela rede de interconexão para serem processados pelas unidades ROP. A rede também direciona solicitações de leitura de memória de textura do SPA para a DRAM e lê dados da DRAM por uma cache de nível 2 de volta ao SPA.

Cluster de textura/processador (TPC)

Cada TPC contém um controlador de geometria, um controlador SM (SMC), dois multiprocessadores streaming (SMs) e uma unidade de textura, como mostra a Figura A.7.2.

O controlador de geometria mapeia o pipeline de vértice gráfico lógico em recirculação nos SMs físicos, direcionando todo atributo de primitivo e vértice e fluxo de topologia no TPC.

O SMC controla múltiplos SMs, arbitrando a unidade de textura compartilhada, caminho de load/store e caminho de E/S. O SMC atende a três cargas de trabalhos gráficos simultaneamente: vértice, geometria e pixel.

A unidade de textura processa uma instrução de textura para um vértice, geometria ou pixel quad, ou quatro threads de cálculo por ciclo. As origens da instrução de textura são

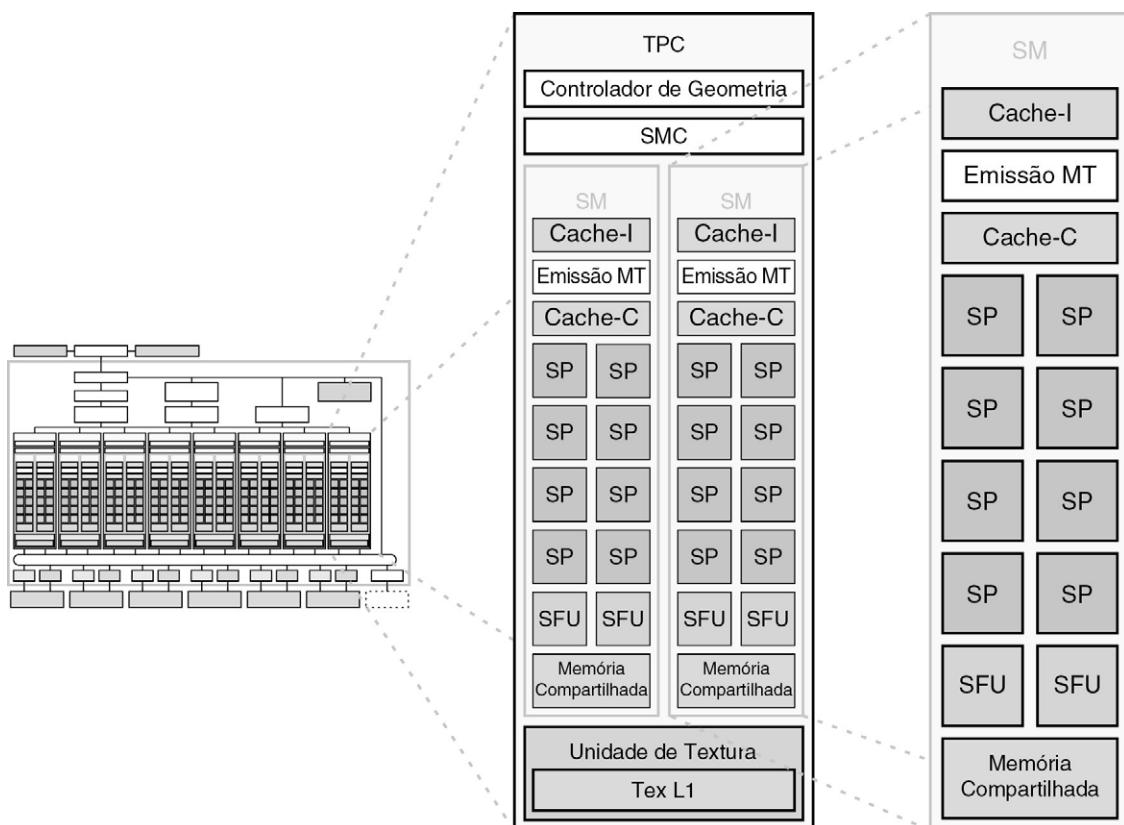


FIGURA A.7.2 Cluster de textura/processador (TPC) e um multiprocessador streaming (SM). Cada SM tem oito cores de processador streaming (SP), duas SFUs e uma memória compartilhada.

coordenadas de textura, e as saídas são amostras ponderadas, normalmente uma cor de ponto flutuante com quatro componentes (RGBA). A unidade de textura utiliza pipelines em profundidade. Embora contenha uma cache streaming para capturar a localidade da filtragem, ela flui acertos misturados com falhas sem gerar stalls.

Multiprocessador streaming (SM)

O SM é um multiprocessador gráfico e de cálculo unificado, que executa programas de sombreamento de vértice, geometria e fragmento de pixel e programas de cálculo paralelos. O SM consiste em oito cores de processador de thread SP, duas SFUs, uma unidade de busca e emissão de instrução multithreaded (emissão MT), uma cache de instrução, uma cache de constante somente de leitura e uma memória compartilhada de leitura/escrita com 16KB. Ele executa instruções escalares para threads individuais.

O clock do GeForce 8800 Ultra trabalha com cores SP e SFUs a 1,5GHz, para um máximo de 36 GFLOPS por SM. Para otimizar a eficiência de potência e área, algumas unidades não de caminho de dados do SM operam com metade da taxa de clock do SP.

Para executar de forma eficiente centenas de threads paralelas enquanto executa vários programas diferentes, o SM é multithreaded por hardware. Ele controla e executa até 768 threads simultâneos no hardware com overhead de escalonamento zero. Cada thread tem seu próprio estado de execução de thread e pode executar um caminho de código independente.

Um warp consiste em até 32 threads do mesmo tipo — vértice, geometria, pixel ou cálculo. O projeto SIMT, anteriormente descrito na Seção A.4, compartilha a unidade de busca e emissão de instrução SM eficientemente por 32 threads, mas requer um warp completo de threads ativas para eficiência de desempenho plena.

O SM escalona e executa diversos tipos de warp simultaneamente. A cada ciclo de emissão, o escalonador seleciona um dos 24 warps para executar uma instrução de warp SIMT. Uma instrução de warp emitida é executada como quatro conjuntos de 8 threads por quatro ciclos de processador. As unidades SP e SFU executam instruções independentemente, e emitindo instruções entre elas em ciclos alternados, o escalonador pode manter ambas totalmente ocupadas. Um scoreboard qualifica cada warp para emitir cada ciclo. O escalonador de instrução prioriza todos os warps prontos e seleciona um com a prioridade mais alta para emissão. A priorização considera o tipo de warp, o tipo de instrução e a “imparcialidade” para todos os warps sendo executados no SM.

O SM executa arrays de threads cooperativos (CTAs) como múltiplos warps simultâneos que acessam uma região de memória compartilhada alocada dinamicamente para o CTA.

Conjunto de instruções

Threads executam instruções escalares, diferente das arquiteturas anteriores de instrução de vetor da GPU. As instruções escalares são mais simples e amigáveis do compilador. As instruções de textura permanecem baseadas em vetor, apanhando um vetor de coordenadas de origem e retornando um vetor de cores filtrado.

O conjunto de instruções baseado em registrador inclui todas as instruções de aritmética de ponto flutuante e inteiro, transcendentais, lógicas, controle de fluxo, load/store de memória e instruções de textura listadas na tabela de instruções PTX da Figura A.4.3. As instruções load/store da memória utilizam endereçamento de byte inteiro com aritmética de endereço de registrador-mais-offset. Para cálculos, as instruções load/store acessam três espaços de memória de leitura-escrita: memória local para dados por thread, privados, temporários; memória compartilhada para dados por CTA de baixa latência, pelas threads do CTA; e memória global para dados compartilhados por todas as threads. Os programas de cálculo utilizam a instrução rápida de sincronização de barreira `bar.sync` para sincronizar threads dentro de um CTA que se comunicam entre si por meio da memória compartilhada e global. As GPUs da arquitetura Tesla mais recente implementam operações de memória atômicas de PTX, que facilita reduções paralelas e gerenciamento paralelo da estrutura de dados.

Processador streaming (SP)

O core SP multithreaded é o processador de threads principal, apresentado na Seção A.4. Seu arquivo de registradores oferece 1024 registradores escalares de 32 bits para até 96 threads (mais threads que o SP de exemplo da [Seção A.4](#)). Suas operações de adição e multiplicação em ponto flutuante são compatíveis com o padrão IEEE 754 para números de PF com precisão simples, incluindo not-a-number (NaN) e infinito. As operações de adição e multiplicação utilizam o arredondamento-para-par-mais-próximo do IEEE como modo de arredondamento padrão. O core SP também implementa todas as instruções PTX de aritmética com inteiros, comparação, conversão e lógicas de 32 e 64 bits. O processador utiliza o pipeline por completo, e a latência é otimizada para balancear atraso e área.

Unidade de função especial (SFU)

A SFU admite cálculo de funções transcendentais e interpolação de atributo planar. Conforme descrevemos na Seção A.6, ela utiliza a interpolação quadrática com base nas aproximações minimax avançadas para aproximar as funções de recíproco, raiz quadrada recíproca, $\log_2 x$, 2^x e sin/cos em um resultado por ciclo. A SFU também admite interpolação de atributo de pixel como coordenadas de cor, profundidade e textura em quatro amostras por ciclo.

Rasterização

Primitivos de geometria dos SMs entram em sua ordem de entrada round-robin original no bloco viewport/clip/setup/raster/zcull. As unidades de viewport e clip cortam os primitivos até o frustum da view e até quaisquer planos de corte do usuário ativados, e depois transformam os vértices em espaço de tela (pixel).

Os primitivos sobreviventes então vão para a unidade de configuração, que gera equações de aresta para o rasterizador. Um estágio de rasterização coarse gera todos os pedaços de pixels que estão pelo menos parcialmente dentro do primitivo. A unidade zcull mantém uma superfície z hierárquica, rejeitando os pedaços de pixels se forem conservadoramente conhecidos como ocultados pelos pixels desenhados anteriormente. A taxa de rejeição é de até 256 pixels por clock. Os pixels que sobrevivem ao zcull vão então para o estágio de rasterização fina, que gera informações de cobertura e valores de profundidade detalhados.

O teste de profundidade e atualização pode ser realizado antes do sombreamento de fragmento, ou depois, dependendo do estado atual. O SMC monta pixels sobreviventes em warps para serem processados por um SM rodando o sombreamento de pixel atual. O SMC, então, envia o pixel sobrevivente e dados associados ao ROP.

Processador de operações de rastreio (ROP) e o sistema de memória

Cada ROP é emparelhado com uma partição de memória específica. Para cada fragmento de pixel emitido por um programa de sombreamento de pixel, os ROPs realizam teste e atualizações de profundidade e estêncil, e, em paralelo, mistura de cores e atualizações. A compactação de cores sem perda (até 8:1) e a compactação de profundidade (até 8:1) são usadas para reduzir a largura de banda da DRAM. Cada ROP tem uma taxa máxima de quatro pixels por clock e admite formatos HDR de ponto flutuante com 16 e 32 bits. ROPs admitem processamento com profundidade de taxa dupla quando as escritas de cor são desativadas.

O suporte para antialiasing inclui até 16 multisampling e supersampling. O algoritmo de antialiasing de amostragem de cobertura (CSAA) calcula e armazena a cobertura Booleana em até 16 amostras e compacta informações redundantes de cor, profundidade e estêncil no footprint da memória e uma largura de banda de quatro ou oito amostras para melhorar o desempenho.

O barramento de dados da memória DRAM é de 384 pinos, organizados em seis partições independentes de 64 pinos cada. Cada partição admite protocolos de taxa de dados dupla DDR2 e GDDR3 orientado a gráficos, em até 1,0GHz, gerando uma largura de banda de cerca de 16GB/s por partição, ou 96GB/s.

Os controladores de memória admitem uma grande faixa de taxas de clock de DRAM, protocolos, densidades de dispositivo e larguras de barramento de dados. Solicitações de textura e load/store podem ocorrer entre qualquer TPC e qualquer partição de memória, de modo que uma rede de interconexão roteia solicitações e respostas.

Escalabilidade

A arquitetura unificada Tesla é projetada por escalabilidade. Variar o número de SMs, TPCs, ROPs, caches e partições de memória oferece o equilíbrio certo para alvos de desempenho e custo nos segmentos de mercado da GPU. A interconexão de link escalável (SLI) conecta múltiplas GPUs, oferecendo mais escalabilidade.

Desempenho

O GeForce 8800 Ultra usa 1,5GHz de clock nos cores do processador de threads SP e SFUs, para um pico de operação teórico de 576 GFLOPS. O GeForce 8800 GTX tem um clock de processador de 1,35GHz e um máximo correspondente de 518 GFLOPS.

As três seções seguintes comparam o desempenho de uma GPU GeForce 8800 com uma CPU multicore em três aplicações diferentes — álgebra linear densa, transformações de Fourier rápidas e classificação. Os programas e bibliotecas da GPU são código C CUDA compilado. O código da CPU utiliza a biblioteca MKL 10.0 da Intel multithreading com precisão simples para aproveitar instruções SSE e múltiplos cores.

Desempenho da álgebra linear densa

Os cálculos de álgebra linear densa são fundamentais em muitas aplicações. Volkov e Demmel [2008] apresentam resultados de desempenho de GPU e CPU para multiplicação matriz-matriz densa com precisão simples (a rotina SGEMM) e fatorações de matriz LU, QR e Cholesky. A Figura A.7.3 compara as taxas em GFLOPS sobre a multiplicação matriz-matriz densa SGEMM para uma GPU GeForce 8800 GTX com uma CPU quad-core. A Figura A.7.4 compara as taxas em GFLOPS sobre a fatoração de matriz para uma GPU com uma CPU quad-core.

Como a multiplicação matriz-matriz SGEMM e as rotinas BLAS3 semelhantes são o centro do trabalho na fatoração de matriz, seu desempenho define um limite superior na taxa de fatoração. Como a ordem de matriz aumenta além de 200 para 400, o problema de fatoração torna-se tão grande que SGEMM pode aproveitar o paralelismo da GPU e contornar o sistema CPU-GPU e overhead de cópia. A multiplicação matriz-matriz SGEMM de Volkov alcança 206 GFLOPS, cerca de 60% da taxa de multiplicação-adição máxima do GeForce 8800 GTX, enquanto a fatoração QR alcançou 192 GFLOPS, cerca de 4,3 vezes a CPU quad-core.

Desempenho de FFT

Fast Fourier Transforms são usadas em muitas aplicações. Grandes transformações e transformações multidimensionais são particionadas em lotes de transformações 1D menores.

A Figura A.7.5 compara o desempenho FFT de precisão simples complexo em 1D de um GeForce 8800 GTX a 1,35GHz (datado de final de 2006) com uma série Intel Xeon E5462 quad-core a 2,8GHz (apelidado de “Harpertown”, datado de final de 2007). O desempenho da CPU era medido usando a Intel Math Kernel Library (MKL) 10.0 FFT com quatro threads. O desempenho da GPU foi medido usando a biblioteca NVIDIA CUFFT 2.1 e FFTs de decimação-em-frequência com raiz 16 1D em lote. O desempenho

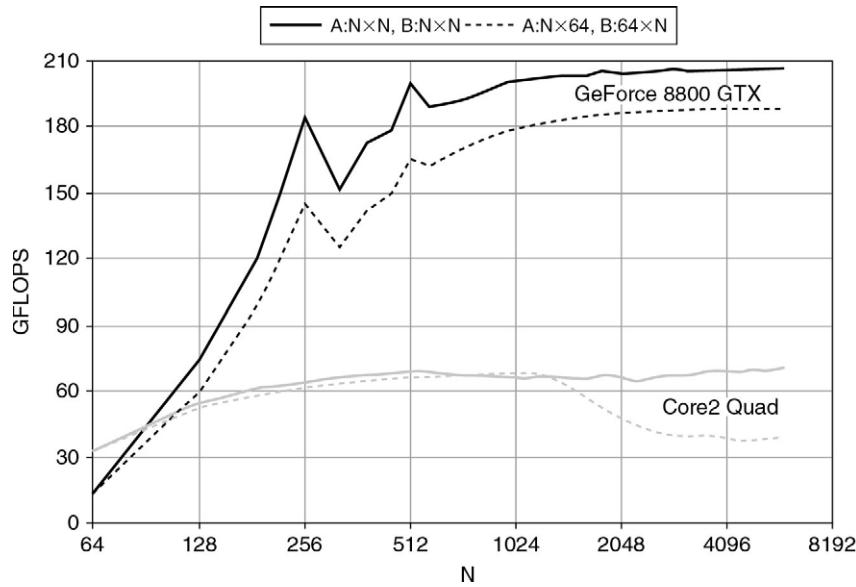


FIGURA A.7.3 Taxas de desempenho da multiplicação matriz-matriz densa SGEMM. O gráfico mostra as taxas em GFLOPS de precisão simples obtidas na multiplicação de matrizes $N \times N$ quadradas (linhas sólidas) e matrizes $N \times 64$ e $64 \times N$ (linhas tracejadas). Adaptado da Figura 6 de Volkov e Demmel [2008]. As linhas pretas são um GeForce 8800 GTX a 1,35GHz usando o código SGEMM de Volkov (agora no NVIDIA CUBLAS 2.0) em matrizes na memória GPU. As linhas azuis são um Intel Core2 Quad Q6600 quad-core a 2,4GHz, Linux com 64 bits, Intel MKL 10.0 em matrizes na memória da CPU.

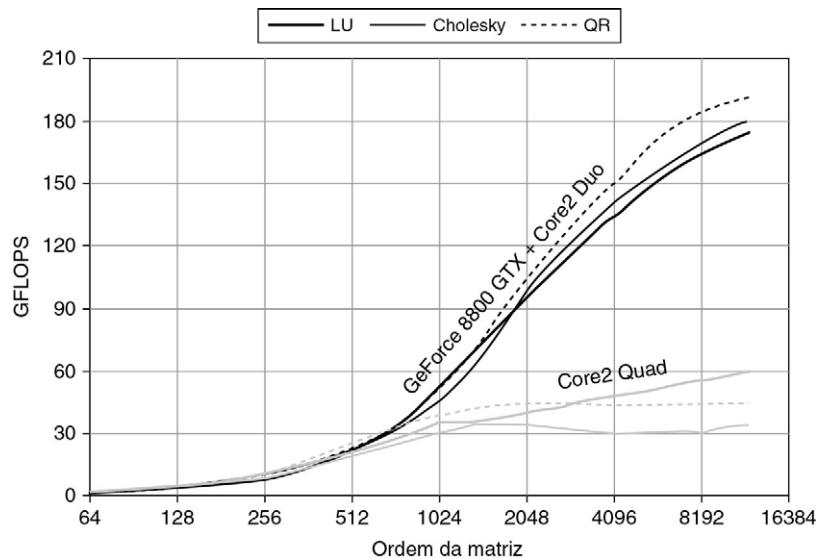


FIGURA A.7.4 Taxas de desempenho de fatoração de matriz densa. O gráfico mostra taxas GFLOPS obtidas em fatorações de matriz usando a GPU e usando apenas a CPU. Adaptado da Figura 7 de Volkov e Demmel [2008]. As linhas pretas são um NVIDIA GeForce 8800 GTX a 1,35GHz, CUDA 1.1, Windows XP conectado a um Intel Core2 Duo E6700 a 2,67GHz, incluindo todos os tempos de transferência de dados CPU-GPU. As linhas azuis são um Intel Core2 Quad Q6600 a 2,4GHz, Linux de 64 bits, Intel MKL 10.0.

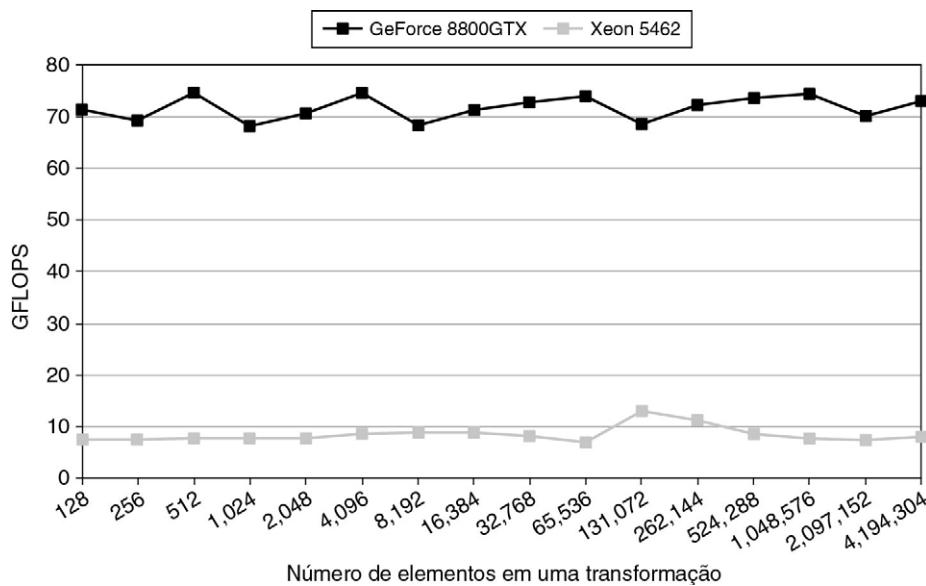


FIGURA A.7.5 Desempenho de vazão Fast Fourier Transform. O gráfico compara o desempenho das FFTs complexas no local unidimensional em lote, em um GeForce 8800 GTX de 1,35GHz com um Intel Xeon E5462 quad-core de 2,8GHz (apelidado “Harpertown”), cache L2 de 6MB, 4GB de memória, 1600 FSB, Red Hat Linux, Intel MKL 10.0.

da vazão da CPU e GPU foi medido usando-se FFTs em lote, com tamanho de lote $2^{24}/n$, onde n é o tamanho da transformação. Assim, a carga de trabalho para cada tamanho de transformação foi 128MB. Para determinar a taxa em GFLOPS, o número de operações por transformação foi tomado como $5n \log_2 n$.

Desempenho da classificação

Ao contrário das aplicações que discutimos, a classificação requer uma coordenação muito mais substancial entre as threads paralelas, e a expansão paralela é correspondentemente mais difícil de obter. Apesar disso, diversos algoritmos de classificação bem conhecidos podem ser eficientemente colocados em paralelo para rodar em na GPU. Satish *et al.* [2008] detalham o projeto de algoritmos de classificação em CUDA, e os resultados que eles informam para radix sort são resumidos a seguir.

A Figura A.7.6 compara o desempenho da classificação paralela de um GeForce 8800 Ultra com um sistema Intel Clovertown de oito cores, ambos datando do início de 2007. Os cores da CPU são distribuídos entre dois soquetes físicos. Cada soquete contém um módulo multichip com chips Core2 gêmeos, e cada chip tem uma cache L2 de 4MB. Todas as rotinas de classificação foram projetadas para classificação pares de chave-valor em que chaves e valores são inteiros de 32 bits. O algoritmo principal sendo estudado é o radix sort, embora o procedimento `parallel_sort()` baseado no Quicksort fornecido pelos Threading Building Blocks da Intel também seja incluído por comparação. Dos dois códigos radix sort baseados na CPU, um foi implementado usando apenas o conjunto de instruções escalares e o outro utiliza rotinas de linguagem assembly cuidadosamente ajustadas, que tiram proveito das instruções de vetor SSE2 SIMD.

O próprio gráfico mostra a taxa de classificação alcançada — definida como o número de elementos classificados dividido pelo tempo a classificar — para um intervalo de tamanhos de sequência. Fica aparente por esse gráfico que a radix sort da GPU alcançou a taxa de classificação mais alta para todas as sequências de elementos de 8K e maiores. Nessa faixa,

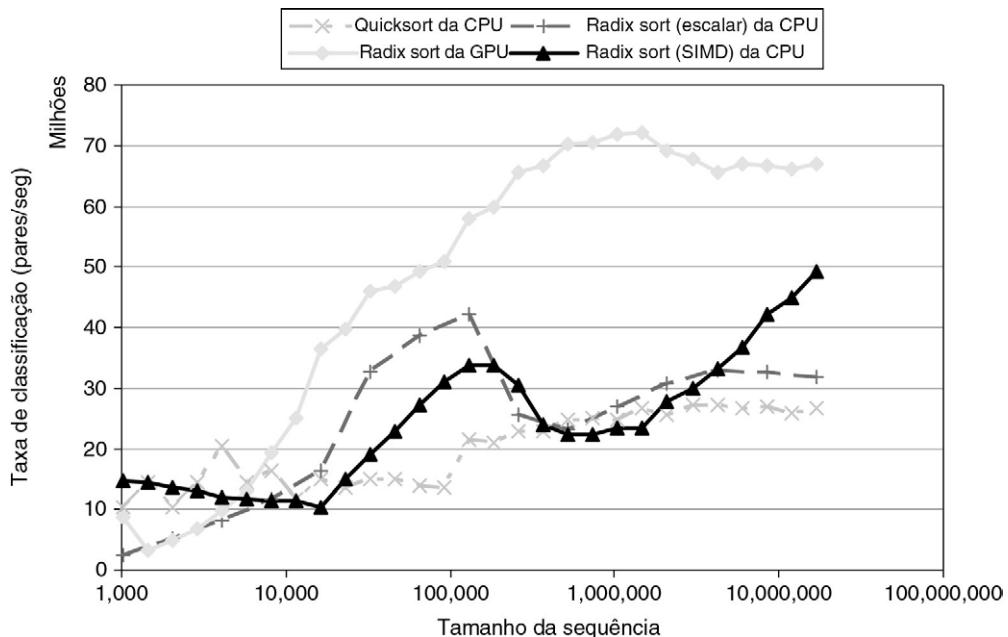


FIGURA A.7.6 Desempenho da classificação paralela. Este gráfico compara as taxas de classificação para implementações de radix sort paralelas em um GeForce 8800 Ultra a 1,5GHz e um sistema de oito cores Intel Core2 Xeon E5345 a 2,33GHz.

ela é na média 2,6 vezes mais rápida que a rotina baseada no Quicksort e aproximadamente duas vezes mais rápida que as rotinas radix sort, todas usando os oito cores da CPU disponíveis. O desempenho da radix sort da CPU varia bastante, provavelmente devido à fraca localidade de cache de suas permutações globais.

A.8

Vida real: mapeando aplicações a GPUs

O advento das CPUs multicore e GPUs manycore significa que os principais chips de processador agora são sistemas paralelos. Além do mais, seu paralelismo continua a se expandir com a lei de Moore. O desafio é desenvolver aplicações importantes de cálculo visual e cálculo de alto desempenho que expandam transparentemente seu paralelismo para aproveitar o número cada vez maior de cores de processador, assim como as aplicações gráficas 3D expandem transparentemente seu paralelismo a GPUs com quantidades bastante variadas de cores.

Esta seção apresenta exemplos de mapeamento de aplicações de cálculo paralelo escaláveis à GPU usando CUDA.

Matrizes esparsas

Uma grande variedade de algoritmos paralelos pode ser escrita em CUDA de uma maneira razoavelmente simples, mesmo quando as estruturas de dados envolvidas não são simples grades regulares. A multiplicação de vetor de matriz esparsa (SpMV) é um bom exemplo de um bloco de montagem importante que pode ser paralelizado diretamente usando as abstrações fornecidas pelo modelo CUDA. Os kernels que discutimos a seguir, quando combinados com as rotinas de vetor CUBLAS fornecidas, simplificam a escrita de solucionadores iterativos, como o método de gradiente conjugado.

Uma matriz esparsa $n \times n$ é aquela em que o número de entradas diferentes de zero m é somente uma pequena fração do total. As representações em matriz esparsa buscam

armazenar apenas os elementos diferentes de zero de uma matriz. Como é muito comum que uma matriz esparsa $n \times n$ contenha apenas $n = O(n)$ elementos diferentes de zero, isso representa uma economia substancial no espaço de armazenamento e tempo de processamento.

Uma das representações mais comuns para as matrizes esparsas não estruturadas gerais é a representação da linha esparsa compactada (CSR). Os m elementos diferentes de zero da matriz A são armazenados em ordem principal de linha em um array Av . Um segundo array Aj registra o índice de coluna correspondente para cada entrada de Av . Finalmente, um array Ap de $n + 1$ elementos registra a extensão de cada linha nos arrays anteriores; as entradas para a linha i em Aj e Av se estendem do índice $Ap[i]$ até, mas não incluindo, o índice $Ap[i+1]$. Isso significa que $Ap[0]$ sempre será 0 e $Ap[n]$ sempre será o número de elementos diferentes de zero na matriz. A [Figura A.8.1](#) mostra um exemplo da representação CSR de uma matriz simples.

Dada uma matriz A em formato CSR e um vetor x , podemos calcular uma única linha do produto $y = Ax$ usando o procedimento `multiply_row()` mostrado na [Figura A.8.2](#). Calcular o produto total é então simplesmente uma questão de percorrer todas as linhas e calcular o resultado para essa linha usando `multiply_row()`, como no código serial em C mostrado na [Figura A.8.3](#).

Esse algoritmo pode ser traduzido para um kernel CUDA paralelo muito facilmente. Simplesmente espalhamos o loop em `csrmul_serial()` por muitas threads paralelas. Cada thread calculará exatamente uma linha do vetor de saída y . O código para esse kernel aparece na [Figura A.8.4](#). Observe que ele se parece exatamente com o loop serial usado no procedimento `csrmul_serial()`. Na realidade, existem apenas dois pontos de diferença. Primeiro, o índice `row` para cada thread é calculado a partir dos índices de bloco e `thread` atribuídos a cada thread, eliminando o loop `for`. Segundo, temos uma condicional que só avalia o produto de uma linha se o índice da linha estiver dentro dos limites da matriz (isso é necessário porque o número de linha n não precisa ser um múltiplo do tamanho de bloco usado na partida do kernel).

$$A = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Av[7] = {	(3 1)	(2 4 1)	(1 1)	}
Aj[7] = {	(0 2)	(1 2 3)	(0 3)	}
Ap[5] = {	0 2	2 5	7	}

a. Matriz de exemplo A b. Representação CSR da matriz

FIGURA A.8.1 Matriz de linha esparsa compactada (CSR).

```

float multiply_row(unsigned int rowsize,
                    unsigned int *Aj, // índices de coluna para linha
                    float *Av,        // entradas não zero para linha
                    float *x)         // o vetor RHS
{
    float sum = 0;

    for(unsigned int column=0; column<rowsize; ++column)
        sum += Av[column] * x[Aj[column]];

    return sum;
}
    
```

FIGURA A.8.2 Código serial em C para uma única linha de multiplicação de vetor de matriz esparsa.

```

void csrmul_serial(unsigned int *Ap, unsigned int *Aj,
                    float *Av, unsigned int num_rows,
                    float *x, float *y)
{
    for(unsigned int row=0; row<num_rows; ++row)
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}

```

FIGURA A.8.3 Código serial para multiplicação de vetor de matriz esparsa.

```

__global__
void csrmul_kernel(unsigned int *Ap, unsigned int *Aj,
                    float *Av, unsigned int num_rows,
                    float *x, float *y)
{
    unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;

    if( row<num_rows )
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin, Aj+row_begin,
                              Av+row_begin, x);
    }
}

```

FIGURA A.8.4 Versão CUDA da multiplicação de vetor de matriz esparsa.

Supondo que as estruturas de dados da matriz já tenham sido copiadas para a memória do dispositivo de GPU, a partida desse kernel se parecerá com isto:

```

unsigned int blocksize = 128; // or any size up to 512
unsigned int nblocks   = (num_rows + blocksize - 1) / blocksize;
csrmul_kernel<<<nblocks,blocksize>>>(Ap, Aj, Av, num_rows, x, y);

```

O padrão que vemos aqui é muito comum. O algoritmo serial original é um loop cujas interações são independentes uma da outra. Esses loops podem ser colocados em paralelo facilmente pela simples atribuição de uma ou mais iterações do loop a cada thread paralela. O modelo de programação fornecido pelo modelo CUDA torna a expressão desse tipo de paralelismo particularmente simples.

Essa estratégia geral de decompor cálculos em blocos de trabalho independente, e mais especificamente desmembrar iterações de loop independentes, não é exclusivo do modelo CUDA. Essa é uma técnica comum utilizada, de uma forma ou de outra, por diversos sistemas de programação paralelos, incluindo OpenMP e Threading Building Blocks da Intel.

Caching na memória compartilhada

Os algoritmos SpMV esboçados aqui são muito simples. Existem diversas otimizações que podem ser feitas nos códigos da CPU e GPU que podem melhorar o desempenho, incluindo desdobramento de loop, reordenação de matriz e bloqueio de registrador. Os kernels paralelos também podem ser reimplementados em termos de operações *scan* paralelas com dados, apresentadas por Sengupta, et al. [2007].

Um dos recursos arquitetônicos importantes expostos pelo modelo CUDA é a presença da memória compartilhada por bloco, uma pequena memória no chip, com latência muito baixa. Tirar proveito dessa memória pode oferecer melhorias de desempenho substanciais. Um modo importante de fazer isso é usar memória compartilhada como uma cache gerenciada pelo software para manter dados reutilizados com frequência. Modificações usando memória compartilhada podem ser vistas na [Figura A.8.5](#).

No contexto da multiplicação de matriz esparsa, observamos que várias linhas de A podem usar um elemento de array em particular $x[i]$. Em muitos casos comuns, e particularmente quando a matriz foi reordenada, as linhas usando $x[i]$ serão linhas próximas da linha i . Portanto, podemos implementar um esquema de caching simples e esperar alcançar algum benefício no desempenho. O bloco de threads processando as linhas de i a j carregará de $x[i]$ a $x[j]$ nessa memória compartilhada. Desdoblaremos o loop `multiply_row()` e buscaremos os elementos de x da cache sempre que for possível. O código resultante aparece na [Figura A.8.5](#). A memória compartilhada também pode ser usada para fazer outras otimizações, como a busca de $Ap[row+1]$ a partir de uma thread adjacente, em vez de buscá-lo novamente da memória.

Como a arquitetura Tesla oferece uma memória compartilhada no chip controlada explicitamente, em vez de uma cache de hardware implicitamente ativa, é muito comum incluir esse tipo de otimização. Embora isso possa impor algum peso de desenvolvimento adicional para o programador, ele é relativamente pequeno, e os benefícios em potencial do desempenho podem ser substanciais. No exemplo mostrado anteriormente, até mesmo esse uso muito simples da memória compartilhada retorna uma melhoria de desempenho de cerca de 20% nas matrizes representativas derivadas de malhas de superfície 3D. A disponibilidade de uma memória controlada explicitamente no lugar de uma cache implícita também tem a vantagem de que políticas de caching e prefetching podem ser ajustadas especificamente às necessidades da aplicação.

Esses são kernels muito simples, cuja finalidade é ilustrar as técnicas básicas na escrita de programas CUDA, em vez de como conseguir o máximo de desempenho. Diversas avenidas de otimização possíveis estão disponíveis, várias delas exploradas por Williams et al. [2007] em algumas arquiteturas multicore diferentes. Apesar disso, ainda é interessante examinar o desempenho comparativo até mesmo desses kernels simples. Em um processador Intel Core2 Xeon E5335 a 2GHz, o kernel `csrmul_serial()` roda em aproximadamente 202 milhões de valores diferentes de zero processados por segundo, para uma coleção de matrizes Laplacianas derivadas de malhas de superfície triangulada 3D. O paralelismo desse kernel com a construção `parallel_for` fornecida pelo Threading Building Blocks da Intel produz speed-ups paralelos de 2,0, 2,1 e 2,3 rodando em dois, quatro e oito cores da máquina, respectivamente. Em um GeForce 8800 Ultra, os kernels `csrmul_kernel()` e `csrmul_cached()` alcançam taxas de processamento de aproximadamente 772 e 920 milhões de valores diferentes de zero por segundo, correspondentes a speed-ups paralelos de 3,8 a 4,6 vezes em relação ao desempenho serial de um único core da CPU.

Varredura e redução

O *scan* paralelo, também conhecido como *soma de prefixo* paralela, é um dos blocos de montagem mais importantes para os algoritmos paralelos de dados [Blelloch, 1990]. Dada uma sequência a e n elementos:

$$[a_0, a_1, \dots, a_{n-1}]$$

```

__global__
void csrmul_cached(unsigned int *Ap, unsigned int *Aj,
                    float *Av, unsigned int num_rows,
                    const float *x, float *y)
{
    // Coloca as linhas de x em [] cache correspondentes a esse bloco.
    __shared__ float cache[blocksize];

    unsigned int block_begin = blockIdx.x * blockDim.x;
    unsigned int block_end   = block_begin + blockDim.x;
    unsigned int row         = block_begin + threadIdx.x;

    // Busca e cache de nossa janela de x[].
    if( row<num_rows) cache[threadIdx.x] = x[row];
    __syncthreads();

    if( row<num_rows )
    {
        unsigned int row_begin = Ap[row];
        unsigned int row_end   = Ap[row+1];
        float sum = 0, x_j;

        for(unsigned int col=row_begin; col<row_end; ++col)
        {
            unsigned int j = Aj[col];

            // Busca x_j da nossa cache quando possível
            if( j>=block_begin && j<block_end )
                x_j = cache[j-block_begin];
            else
                x_j = x[j];

            sum += Av[col] * x_j;
        }

        y[row] = sum;
    }
}

```

FIGURA A.8.5 Versão de memória compartilhada da multiplicação de vetor por matriz esparsa.

e um operador associativo binário \oplus , a função `scan` calcula a sequência:

$$\text{scan}(a, \oplus) = [a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-1})]$$

Como exemplo, se considerarmos \oplus como o operador de adição normal, então aplicar `scan` ao array de entrada

$$a = [3 1 7 0 4 1 6 3]$$

produzirá esta sequência de somas parciais:

$$\text{scan}(a, +) = [3 4 11 11 15 16 22 25]$$

Essa operação de scan é um scan *inclusivo*, no sentido de que o elemento i da sequência de saída incorpora o elemento a_i da entrada. A incorporação apenas dos elementos anteriores geraria um operador de scan *exclusivo*, também conhecido como operação de *soma de prefixo*.

A implementação serial dessa operação é extremamente simples. Ela é simplesmente um loop que se repete uma vez pela sequência inteira, como mostra a [Figura A.8.6](#).

À primeira vista, pode parecer que essa operação é inherentemente serial. Porém, ela pode realmente ser implementada em paralelo eficientemente. A principal observação é que, como a adição é associativa, estamos livres para mudar a ordem em que os elementos são somados. Por exemplo, podemos imaginar a soma de pares de elementos consecutivos em paralelo, e depois a soma dessas somas parciais, e assim por diante.

Um esquema simples para fazer isso vem de Hillis e Steele [1989]. Uma implementação de seu algoritmo em CUDA aparece na [Figura A.8.7](#). Ela considera que o array de entrada $x[]$ terá exatamente um elemento por thread do bloco de threads. Ela realiza $\log_2 n$ iterações de um loop coletando somas parciais.

Para entender a ação desse loop, considere a [Figura A.8.8](#), que ilustra o caso simples para $n = 8$ threads e elementos. Cada nível do diagrama representa um passo do loop. As linhas indicam o local do qual os dados estão sendo buscados. Para cada elemento da saída (ou seja, a linha final do diagrama), estamos montando uma árvore de somatório pelos elementos da entrada. As arestas destacadas em azul mostraram a forma dessa árvore de somatório para o elemento final. As folhas dessa árvore são todas elementos iniciais. Voltando a partir de qualquer elemento da saída, vemos que ele incorpora todos os valores de entrada até ele mesmo, inclusive.

Embora simples, esse algoritmo não é tão eficiente quanto gostaríamos. Examinando a implementação serial, vemos que ela realiza $O(n)$ adições. A implementação paralela, ao

```
template<class T>
__host__ T plus_scan(T *x, unsigned int n)
{
    for(unsigned int i=1; i<n; ++i)
        x[i] = x[i-1] + x[i];
}
```

FIGURA A.8.6 Modelo para plus-scan serial.

```
template<class T>
__device__ T plus_scan(T *x)
{
    unsigned int i = threadIdx.x;
    unsigned int n = blockDim.x;

    for(unsigned int offset=1; offset<n; offset *= 2)
    {
        T t;

        if(i>=offset) t = x[i-offset];
        __syncthreads();

        if(i>=offset) x[i] = t + x[i];
        __syncthreads();
    }
    return x[i];
}
```

FIGURA A.8.7 Modelo CUDA para plus-scan paralelo.

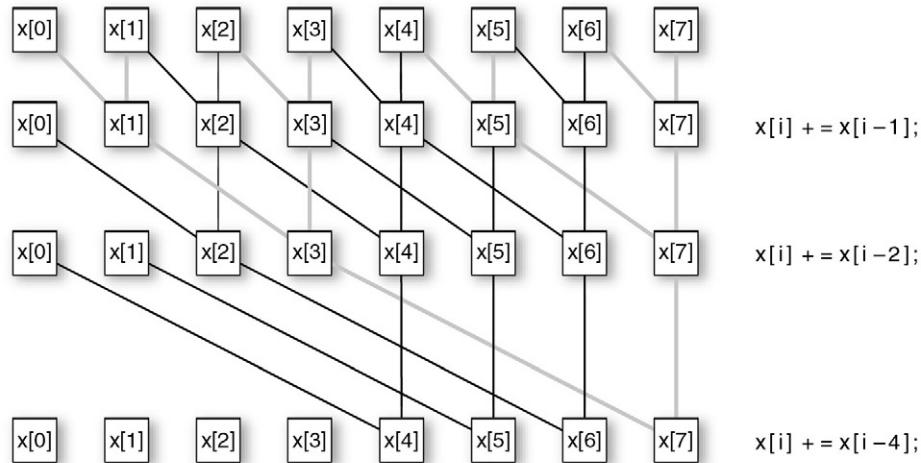


FIGURA A.8.8 Referências de dados de varredura paralelos baseados em árvore.

contrário, realiza $O(n \log n)$ adições. Por esse motivo, ela não é eficiente ao trabalho, pois realiza mais trabalho que a implementação serial para calcular o mesmo resultado. Felizmente, existem outras técnicas para implementar `scan` que são eficientes ao trabalho. Os detalhes sobre técnicas de implementação mais eficientes e a extensão desse procedimento por bloco a arrays multiblocos são fornecidos por Sengupta, et al. [2007].

Em alguns casos, podemos só estar interessados em calcular a soma de todos os elementos em um array, em vez de uma sequência de todas as somas de prefixo retornadas por `scan`. Esse é o problema da *redução paralela*. Poderíamos simplesmente usar um algoritmo de `scan` para realizar esse cálculo, mas a redução geralmente pode ser implementada de forma mais eficiente que o `scan`.

A Figura A.8.9 mostra o código para calcular uma redução usando a adição. Nesse exemplo, cada thread simplesmente carrega um elemento da sequência de entrada (ou seja, ela inicialmente soma uma subseqüência de comprimento 1). Ao final da redução, queremos que a thread 0 contenha a soma de todos os elementos carregados inicialmente pelas threads de seu bloco. O loop nesse kernel monta implicitamente uma árvore de somatório sobre os elementos de entrada, de modo semelhante ao algoritmo de `scan` acima.

Ao final deste loop, a thread 0 contém a soma de todos os valores carregados por esse bloco. Se quisermos que o valor final do local apontado por `total` contenha o total de todos os elementos no array, temos de combinar as somas parciais de todos os blocos na grade. Uma estratégia para seria fazer com que cada globo escreva sua soma parcial em um segundo array e depois iniciar o kernel de redução novamente, repetindo o processo até que tenhamos reduzido a sequência a um único valor. Uma alternativa mais atraente admitida pela arquitetura de GPU Tesla é usar o primitivo `atomicAdd()`, um primitivo de leitura-modificação-escrita atômico admitido pelo subsistema de memória. Isso elimina a necessidade de arrays temporários adicionais e partidas de kernel repetidas.

A redução paralela é um primitivo essencial para a programação paralela e realça a importância da memória compartilhada por bloco e barreiras de baixo custo para tornar eficiente a cooperação entre as threads. Esse grau de mistura de dados entre as threads seria proibitivamente caro se fosse feito na memória global fora do chip.

Radix sort

Uma aplicação importante das primitivas de `scan` é na implementação de rotinas de classificação. O código na Figura A.8.10 implementa um radix sort de inteiros por um único bloco de threads. Ele aceita como entrada um array `values` contendo um inteiro de 32 bits para cada thread do bloco. Por eficiência, esse array deverá ser armazenado na

```

__global__
void plus_reduce(int *input, unsigned int N, int *total)
{
    unsigned int tid = threadIdx.x;
    unsigned int i   = blockIdx.x*blockDim.x + threadIdx.x;

    // Cada bloco carrega seus elementos na memória compartilhada,
    // preenchendo com 0 se N não for um múltiplo de blocksize
    __shared__ int x[blocksize];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    // Cada thread agora mantém 1 valor de entrada em x[]
    //
    // Monta árvore de resumo pelos elementos.
    for(int s=blockDim.x/2; s>0; s=s/2)
    {
        if(tid < s) x[tid] += x[tid + s];
        __syncthreads();
    }

    // Thread 0 agora contém a soma de todos os valores de entrada
    // para esse bloco. Se tivesse somado essa soma no total acumulado
    if( tid == 0 ) atomicAdd(total, x[tid]);
}

```

FIGURA A.8.9 Implementação CUDA de plus-reduction.

```

__device__ void radix_sort(unsigned int *values)
{
    for(int bit=0; bit<32; ++bit)
    {
        partition_by_bit(values, bit);
        __syncthreads();
    }
}

```

FIGURA A.8.10 Código CUDA para radix sort.

memória compartilhada por bloco, mas isso não é exigido para que a classificação ocorra corretamente.

Essa é uma implementação muito simples do radix sort. Ela considera a disponibilidade de um procedimento `partition_by_bit()` que dividirá o array indicado de modo que todos os valores com um 0 no bit designado chegarão antes de todos os valores com 1 nesse bit. Para produzir a saída correta, esse particionamento deverá ser estável.

A implementação de um procedimento de particionamento é uma aplicação simples do scan. A thread i mantém o valor x_i e precisa calcular o índice de saída correto para escrever esse valor. Para fazer isso, ela precisa calcular (1) o número de threads $j < i$ para as quais o bit designado é 1 e (2) o número total de bits para os quais o bit designado é 0. O código CUDA para `partition_by_bit()` aparece na Figura A.8.11.

Uma estratégia semelhante pode ser aplicada para implementar um kernel de radix sort que classifica um array de tamanho grande, em vez de apenas um array de um bloco. A

```

__device__ void partition_by_bit(unsigned int *values,
                                unsigned int bit)
{
    unsigned int i      = threadIdx.x;
    unsigned int size = blockDim.x;
    unsigned int x_i   = values[i];
    unsigned int p_i   = (x_i >> bit) & 1;

    values[i] = p_i;
    __syncthreads();

    // Compute number of T bits up to and including p_i.
    // Record the total number of F bits as well.
    unsigned int T_before = plus_scan(values);
    unsigned int T_total  = values[size-1];
    unsigned int F_total  = size - T_total;
    __syncthreads();

    // Escreve cada x_i em seu próprio lugar
    if( p_i )
        values[T_before-1 + F_total] = x_i;
    else
        values[i - T_before] = x_i;
}

```

FIGURA A.8.11 Código CUDA para particionar dados com base em cada bit, como parte do radix sort.

etapa fundamental continua sendo o procedimento de scan, embora, quando um cálculo é particionado por múltiplos kernels, devamos dobrar o buffer do array de valores em vez de fazer o particionamento no local. Satish, Harris e Garland [2008] fornecem detalhes para a realização de radix sorts sobre arrays grandes de forma eficiente.

Aplicações N-body em uma GPU¹

Nyland, Harris e Prins [2007] descrevem um kernel de cálculo simples, porém útil, com excelente desempenho de GPU — o algoritmo *all-pairs N-body*. Esse é um componente demorado de muitas aplicações científicas. Simulações N-body calculam a evolução de um sistema de corpos em que cada corpo interage continuamente com cada outro corpo. Um exemplo é uma simulação astrofísica em que cada corpo representa uma estrela individual, e os corpos atraem gravitacionalmente uns aos outros. Outros exemplos são desdobramento de proteína, em que a simulação N-body é usada para calcular forças eletrostáticas e van der Waals; simulação de fluxo de fluido turbulento; e iluminação global em gráficos de computador.

O algoritmo all-pairs N-body calcula a força total em cada corpo no sistema pelo cálculo da força de cada par no sistema, somando para cada corpo. Muitos cientistas consideram esse método como o mais preciso, com a única perda de precisão vindo das operações de hardware de ponto flutuante. A desvantagem é sua complexidade de cálculo $O(n^2)$, que é muito grande para os sistemas com mais de 10^6 corpos. Para contornar esse custo alto, várias simplificações foram propostas na geração de algoritmos $O(n \log n)$ e $O(n)$;

¹ Adaptado de Nyland, Harris e Prins [2007], “Fast N-Body Simulation with CUDA”, Capítulo 31 de *GPU Gems 3*.

alguns exemplos são o algoritmo de Barnes-Hut, o Fast Multipole Method e o somatório Particle-Mesh-Ewald. Todos os métodos *rápidos* ainda contam com o método all-pais como um kernel para o cálculo preciso de forças de curta duração; assim, ele continua a ser importante.

Matemática N-body

Para a simulação gravitacional, calcule a força corpo-corpo usando a física elementar. Entre dois corpos indexados por i e j , o vetor força 3D é:

$$f_{ij} = G \frac{m_i m_j}{\|r_{ij}\|^2} \times \frac{r_{ij}}{\|r_{ij}\|}$$

A magnitude da força é calculada no termo da esquerda, enquanto a direção é calculada à direita (vetor unidade apontando de um corpo para o outro).

Dada uma lista de corpos interagindo (um sistema inteiro ou um subconjunto), o cálculo é simples: para todos os pares de interações, calcule a força e a soma para cada corpo. Quando as forças totais são calculadas, elas são usadas para atualizar a posição e a velocidade de cada corpo, com base na posição e velocidade anterior. O cálculo das forças tem complexidade $O(n^2)$, enquanto a atualização é $O(n)$.

O código do cálculo de força serial usa dois loops for aninhados interagindo por pares de corpos. O loop externo seleciona o corpo para o qual a força total está sendo calculada, e o loop interno percorre todos os corpos. O loop interno chama uma função que calcula a força por cada par, depois soma a força a uma soma acumulada.

Para calcular as forças em paralelo, atribuímos uma thread a cada corpo, pois o cálculo da força sobre cada corpo é independente do cálculo em todos os outros corpos. Quando todas as forças são calculadas, as posições e velocidades dos corpos podem ser atualizadas.

O código para as versões serial e paralela aparece na Figura A.8.12 e na Figura A.8.13. A versão serial tem dois loops for aninhados. A conversão para CUDA, como em muitos outros exemplos, converte o loop externo serial para um kernel por thread, em que cada thread calcula a força total sobre um único corpo. O kernel CUDA calcula uma ID de thread global para cada thread, substituindo a variável repetidora do loop externo serial. Os dois kernels terminam armazenando a aceleração total em um array global usado para calcular os novos valores de posição e velocidade em uma etapa subsequente. O loop externo é substituído por uma grade de kernel CUDA que dispara N threads, um para cada corpo.

Otimização para execução da GPU

O código CUDA mostrado está funcionalmente correto, mas não é eficiente, pois ignora recursos arquitetônicos importantes. O melhor desempenho pode ser alcançado com três otimizações principais. Primeiro, a memória compartilhada pode ser usada para evitar

```
void accel_on_all_bodies()
{
    int i, j;
    float3 acc(0.0f, 0.0f, 0.0f);

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            acc = body_body_interaction(acc, body[i], body[j]);
        }
        accel[i] = acc;
    }
}
```

FIGURA A.8.12 Código serial para comparar todas as forças por par em N corpos.

```

__global__ void accel_on_one_body()
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j;
    float3 acc(0.0f, 0.0f, 0.0f);

    for (j = 0; j < N; j++) {
        acc = body_body_interaction(acc, body[i], body[j]);
    }
    accel[i] = acc;
}

```

FIGURA A.8.13 Código de thread CUDA para calcular a força total em um único corpo.

leituras de memória idênticas entre as threads. Segundo, o uso de múltiplas threads por corpo melhora o desempenho para valores pequenos de N . Terceiro, o desdobramento de loop reduz o overhead com o loop.

Usando memória compartilhada

A memória compartilhada pode manter um subconjunto de posições de corpo, como uma cache, eliminando solicitações de memória redundantes entre as threads. Otimizamos o código mostrado anteriormente para que cada uma das p threads em um bloco de threads carregue *uma* posição na memória compartilhada (para um total de p posições). Quando todas as threads tiverem carregado um valor na memória compartilhada, garantido por `_syncthreads()`, cada thread pode então realizar p interações (usando os dados na memória compartilhada). Isso é repetido N/p vezes para completar o cálculo de força para cada corpo, o que reduz o número de solicitações à memória por um fator de p (normalmente, na faixa de 32 a 128).

A função chamada `accel_on_one_body()` requer algumas mudanças para admitir essa otimização. O código modificado aparece na [Figura A.8.14](#).

O loop que anteriormente percorria todos os corpos agora salta pela dimensão de bloco p . Cada iteração do loop externo carrega p posições sucessivas na memória compartilhada (uma posição por thread). As threads se sincronizam e depois p cálculos da força são calculados por cada thread. Uma segunda sincronização é necessária para garantir que novos valores não sejam carregados na memória compartilhada antes de todas as threads, completando os cálculos de força com os dados atuais.

O uso da memória compartilhada reduz a largura de banda de memória exigida para menos de 10% da largura de banda total que a GPU pode sustentar (usando menos de 5GB/s). Essa otimização mantém a aplicação ocupada realizando cálculo, em vez de esperar os acessos à memória, como aconteceria sem o uso da memória compartilhada. O desempenho para valores variados de N aparece na [Figura A.8.15](#).

Usando múltiplas threads por corpo

A [Figura A.8.15](#) mostra a degradação de desempenho para problema com pequenos valores de N ($N < 4096$) no GeForce 8800 GTX. Muitos esforços de pesquisa que contam com cálculos N-body focalizam em um N pequeno (para longos tempos de simulação), tornando-o um alvo para nossos esforços de otimização. Nossa suposição para explicar o desempenho mais baixo foi que simplesmente não havia trabalho suficiente para manter a GPU ocupada quando N é pequeno. A solução é alocar mais threads por corpo. Mudamos as dimensões do bloco de threads de $(p, 1, 1)$ para $(p, q, 1)$, em que q threads dividem o trabalho de um único corpo em partes iguais. Alocando as threads adicionais dentro do mesmo bloco de threads, resultados parciais podem ser armazenados na memória compartilhada. Quando todos os cálculos de força forem feitos, os q resultados parciais podem ser coletados e somados para calcular o resultado final. O uso de duas ou quatro threads por corpo leva a grandes melhorias para um N pequeno.

Como um exemplo, o desempenho no 8800 GTX salta por 110% quando $N = 1024$ (uma thread alcança 90 GFLOPS, em que quatro alcançam 190 GFLOPS). O desempenho cai

```

__shared__ float4 shPosition[256];
...
__global__ void accel_on_one_body()
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j, k;
    int p = blockDim.x;
    float3 acc(0.0f, 0.0f, 0.0f);
    float4 myBody = body[i];

    for (j = 0; j < N; j += p) { // Outer loops jumps by p each time
        shPosition[threadIdx.x] = body[j+threadIdx.x];
        __syncthreads();
        for (k = 0; k < p; k++) { // Inner loop accesses p positions
            acc = body_body_interaction(acc, myBody, shPosition[k]);
        }
        __syncthreads();
    }
    accel[i] = acc;
}

```

FIGURA A.8.14 Código CUDA para calcular a força total em cada corpo, usando memória compartilhada para melhorar o desempenho.

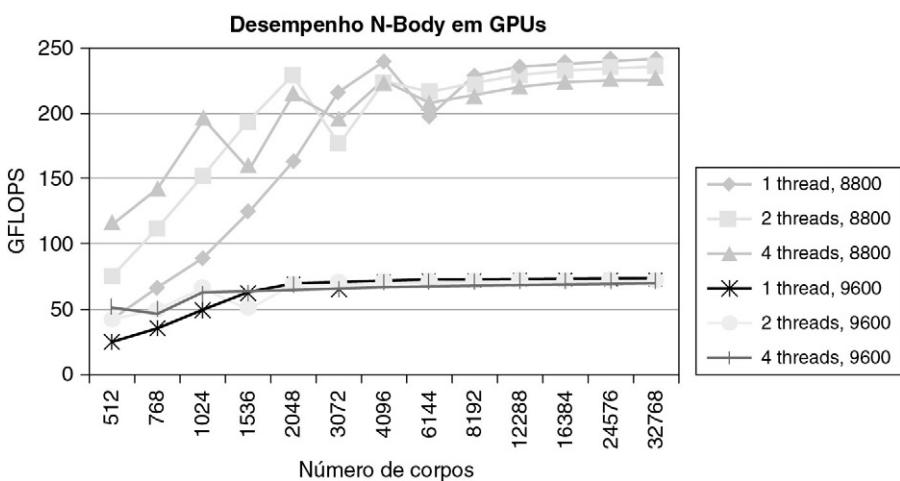


FIGURA A.8.15 Medidas de desempenho da aplicação N-body em um GeForce 8800 GTX e um GeForce 9600. O 8800 tem 128 processadores de stream a 1,35GHz, enquanto o 9600 tem 64 a 0,80GHz (cerca de 30% do 8800). O desempenho de pico é 242 GFLOPS. Para uma GPU com mais processadores, o problema precisa ser maior para alcançar desempenho pleno (o pico de 9600 fica em torno de 2048 corpos, enquanto o 8800 não alcança seu máximo até 16.384 corpos). Para um N pequeno, mais de uma thread por corpo pode melhorar o desempenho significativamente, mas por fim contrai uma penalidade no desempenho à medida que N cresce.

ligeiramente em um N grande, de modo que só usamos essa otimização para N menores que 4096. Os aumentos de desempenho aparecem na Figura A.8.15 para uma GPU com 128 processadores e uma GPU menor com 64 processadores com clock a dois terços da velocidade.

Comparação de desempenho

O desempenho do código N-body aparece nas Figuras A.8.15 e A.8.16. Na Figura A.8.15, podemos ver o desempenho das GPUs como alto e médio, junto com as melhorias alcançadas usando múltiplas threads por corpo. O desempenho na GPU mais rápida varia de 90 a pouco menos de 250 GFLOPS.

A Figura A.8.16 mostra um código quase idêntico ($C++$ versus CUDA) rodando em CPUs Core2 da Intel. O desempenho da CPU é cerca de 1% da GPU, na faixa de 0,2 a 2 GFLOPS, permanecendo quase constante pela grande gama de tamanhos de problema.

O gráfico também mostra os resultados da compilação da versão CUDA do código para uma CPU, em que o desempenho melhora em 24%. CUDA, como linguagem de programação, expõe paralelismo, permitindo que o compilador faça melhor uso da unidade de vetor SSE em um único core. A versão CUDA do código N-body também é naturalmente mapeada para CPUs multicore (com grades de blocos), nas quais alcança expansão quase perfeita em um sistema de oito cores com $N = 4096$ (razões de 2,0, 3,97 e 7,94 em dois, quatro e oito cores, respectivamente).

Resultados

Com um esforço modesto, desenvolvemos um kernel de cálculo que melhora o desempenho da GPU por CPUs multicore por um fator de até 157. O tempo de execução para o código N-body rodando em uma CPU recente da Intel (Penryn X9775 a 3.2 GHz, único core) levou mais de 3 segundos por frame para rodar o mesmo código que roda a uma taxa de frame de 44 Hz em uma GPU GeForce 8800. Em CPUs anteriores ao Penryn, o código exige 6-16 segundos, e nos processadores Core2 mais antigos e no processador Pentium IV, o tempo é de cerca de 25 segundos. Temos de dividir o aumento aparente no desempenho ao meio, pois a CPU requer apenas metade dos cálculos para gerar o mesmo resultado (usando a otimização de que as forças em um par de corpos são iguais em força e opostas em direção).

Como a GPU pode agilizar o código tanto assim? A resposta exige uma inspeção nos detalhes arquitetônicos. O cálculo da força por par requer 20 operações de ponto flutuante, compreendendo principalmente instruções de adição e multiplicação (algumas das quais podem ser combinadas usando-se uma instrução de multiplicação-adição), mas também existem instruções de divisão e raiz quadrada para normalização de vetor. As CPUs da Intel utilizam muitos ciclos para instruções de divisão e raiz quadrada com precisão simples,²

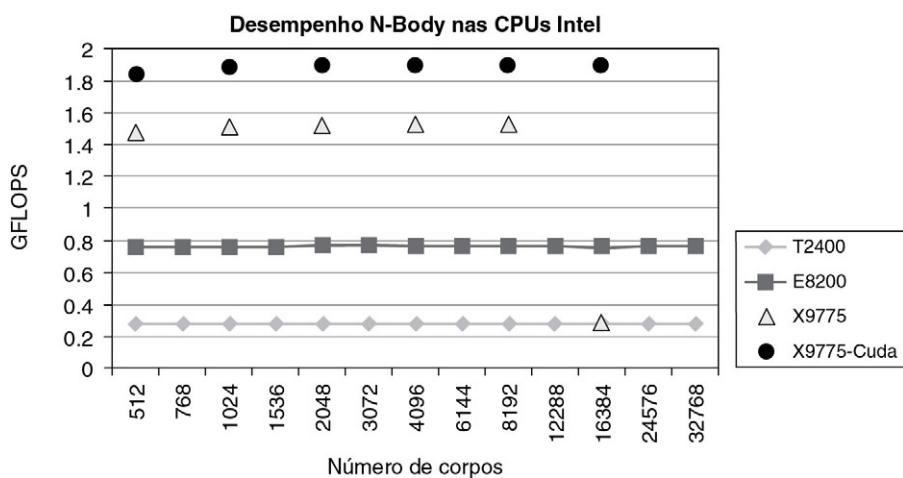


FIGURA A.8.16 Medidas de desempenho no código N-body em uma CPU. O gráfico mostra o desempenho N-body de precisão simples usando CPUs Core2 da Intel, indicados por seu número de modelo de CPU. Observe a drástica redução no desempenho em GFLOPS (mostrado em GFLOPS no eixo y), demonstrando o quanto mais rápida é a GPU em comparação com a CPU. O desempenho na CPU geralmente independe do tamanho do problema, exceto para um desempenho anormalmente baixo quando $N = 16,384$ na CPU X9775. O gráfico também mostra os resultados da execução da versão CUDA do código (usando o compilador CUDA-for-CPU) em um único core da CPU, em que ele supera o código $C++$ em 24%. Como uma linguagem de programação, CUDA expõe paralelismo e localidade, que um compilador pode explorar. As CPUs da Intel são um Extreme X9775 a 3,2GHz (apelidado de “Penryn”), um E8200 a 2,66GHz (apelidado de “Wolfdale”), um desktop, CPU anterior ao Penryn, e um T2400 a 1,83GHz (apelidado de “Yonah”), uma CPU de laptop de 2007. A versão Penryn da arquitetura Core 2 é particularmente interessante para cálculos de N-body com seu divisor de 4 bits, permitindo que operações de divisão e raiz quadrada sejam executadas quatro vezes mais rápido que as CPUs Intel anteriores.

² As instruções SSE do x86 de raiz quadrada recíproca (RSQRT*) e recíproca (RCP*) não foram consideradas, pois sua precisão é muito baixa para ser comparável.

embora isso tenha melhorado na família de CPUs Penryn mais recente, com seu divisor de 4 bits mais rápido.³ Além do mais, as limitações na capacidade de registrador levam a muitas instruções MOV no código x86 (supostamente de/para a cache L1). Ao contrário, o GeForce 8800 executa uma instrução de thread de raiz quadrada recíproca em quatro clocks; veja na Seção A.6 a precisão da função especial. Ele tem um arquivo de registradores maior (por thread) e memória compartilhada que pode ser acessada como um operando de instrução. Finalmente, o compilador CUDA emite 15 instruções para uma interação do loop, em comparação com mais de 40 instruções de uma série de compiladores de CPU x86. Maior paralelismo, execução mais rápida de instruções complexas, mais espaço de registrador e um compilador eficiente se combinam para explicar a melhoria de desempenho dramático do código N-body entre a CPU e a GPU.

Em um GeForce 8800, o algoritmo all-pairs N-body emite mais de 240 GFLOPS de desempenho, em comparação com menos de 2 GFLOPS nos processadores sequenciais recentes. A compilação e execução da versão CUDA do código em uma CPU demonstra que o programa se expande bem para CPUs multicore, mas ainda é significativamente mais lento que uma única GPU.

Acoplamos a simulação N-body da GPU com uma exibição gráfica do movimento, e podemos exibir interativamente 16K corpos interagindo a 44 frames por segundo. Isso permite que eventos astrofísicos e biofísicos sejam exibidos e navegados em taxas interativas. Além disso, podemos parametrizar muitas configurações, como redução de ruído, damping e técnicas de integração, exibindo imediatamente seus efeitos sobre a dinâmica do sistema. Isso oferece aos cientistas uma imagem visual incrível, melhorando suas percepções sobre sistemas de outra forma invisíveis (muito grandes ou pequenos, muito rápidos ou muito lentos), permitindo-lhes criar melhores modelos dos fenômenos físicos.

A Figura A.8.17 mostra uma exibição de série de tempo de uma simulação astrofísica de 16K corpos, com cada corpo atuando como uma galáxia. A configuração inicial é uma

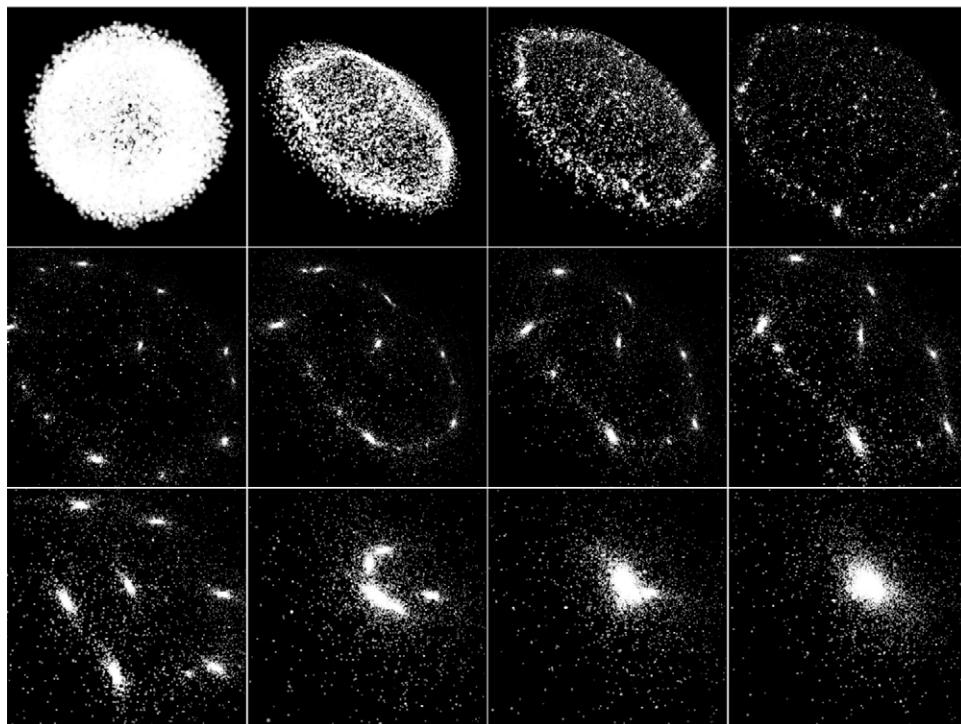


FIGURA A.8.17 Doze imagens capturadas durante a evolução de um sistema N-body com 16.384 corpos.

³ Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Novembro de 2007. Order Number: 248966-016. Também disponível em www3.intel.com/design/processor/manuals/248966.pdf.

concha esférica de corpos girando em torno do eixo z. Um fenômeno de interesse para os astrofísicos é o agrupamento que ocorre, juntamente com a mistura de galáxias com o tempo. Para o leitor interessado, o código CUDA para essa aplicação está disponível no SDK CUDA, em www.nvidia.com/CUDA.

A.9

Falácias e armadilhas

As GPUs evoluíram e mudaram tão rapidamente que surgiram muitas falácias e armadilhas. Abordamos algumas delas aqui.

Falácia: GPUs são apenas multiprocessadores de vetor SIMD. É fácil chegar à falsa conclusão de que as GPUs são simplesmente multiprocessador de vetor SIMD. GPUs possuem um modelo de programação em estilo SPMD, pois um programador pode escrever um único programa que é executado em múltiplas instâncias de thread com múltiplos dados. Contudo, a execução dessas threads não é puramente SIMD ou vetor; ela é Single-Instruction Multiple-Thread (SIMT), descrito na Seção A.4. Cada thread da GPU tem seus próprios registradores escalares, memória particular da thread, estado de execução da thread, ID de thread, execução independente, caminho de desvio e contador de programa eficaz, além de poder endereçar a memória independentemente. Embora um grupo de threads (por exemplo, um warp de 32 threads) seja executado mais eficientemente quando os PCs para as threads são os mesmos, isso não é necessário. Assim, os multiprocessadores não são puramente SIMD. O modelo de execução de thread é MIMD com sincronização de barreira e otimizações SIMT. A execução é mais eficiente se os acessos individuais de load/store da thread também puderem ser reunidos em acessos em bloco. Porém, isso não é estritamente necessário. Em uma arquitetura de vetor puramente SIMD, os acessos à memória/registrador para diferentes threads precisam ser alinhados em um padrão de vetor regular. Uma GPU não possui tal restrição para acessos a registrador ou memória; porém, a execução é mais eficiente se os warps de threads acessarem blocos de dados locais.

Saindo ainda mais de um modelo SIMD puro, uma GPU SIMT pode executar mais de um warp de threads simultaneamente. Em aplicações gráficas, pode haver vários grupos de programas de vértice, programas de pixel e programas de geometria rodando no array multiprocessador simultaneamente. Os programas de cálculo também podem executar diferentes programas simultaneamente em diferentes warps.

Falácia: o desempenho da GPU não pode ficar mais rápido do que a lei de Moore. A lei de Moore é simplesmente uma taxa. Ela não é um limite da “velocidade da luz” para qualquer outra taxa. A lei de Moore descreve uma expectativa de que, com o tempo, à medida que a tecnologia de semicondutores avança e os transistores se tornam menores, o custo de manufatura por transistor cairá exponencialmente. Em outras palavras, dado um custo de manufatura constante, o número de transistores aumentará exponencialmente. Gordon Moore [1965] previu que essa progressão ofereceria aproximadamente duas vezes o número de transistores para o mesmo custo de manufatura a cada ano, e mais tarde o revisou para dobrar a cada dois anos. Embora Moore fizesse a previsão inicial em 1965, quando havia apenas 50 componentes por circuito integrado, ele provou ser muito coerente. A redução do tamanho do transistor historicamente teve outros benefícios, como menor potência por transistor e velocidades de clock mais rápidas em potência constante.

Essa generosidade crescente de transistores é usada por arquiteturas de chip para criar processadores, memória e outros componentes. Por algum tempo, os projetistas de CPU usaram os transistores extras para aumentar o desempenho do processador em uma taxa semelhante à lei de Moore, tanto que muitas pessoas acham que o crescimento do desempenho do processador de duas vezes a cada 18-24 meses é a lei de Moore. Na verdade, não é.

Os projetistas de microprocessador gastam alguns dos novos transistores nos cores do processador, melhorando a arquitetura e o projeto, e realizando o pipelining para mais

velocidade de clock. O restante dos novos transistores é usado para oferecer mais cache e tornar o acesso à memória mais rápido. Ao contrário, os projetistas de GPU utilizam quase nenhum dos novos transistores para oferecer mais cache; a maioria dos transistores é usada para melhorar os cores do processador e acrescentar mais cores do processador.

As GPUs ficam mais rápidas por quatro mecanismos. Primeiro, os projetistas de GPU colhem os frutos da lei de Moore diretamente pela aplicação de exponencialmente mais transistores na montagem de processadores mais paralelos e, portanto, mais rápidos. Segundo, os projetistas de GPU podem melhorar a arquitetura com o tempo, aumentando a eficiência do processamento. Terceiro, a lei de Moore considera custo constante, de modo que a taxa da lei de Moore claramente pode ser excedida gastando-se mais para chips maiores com mais transistores. Quarto, os sistemas de memória da GPU aumentaram sua largura de banda efetiva em um ritmo quase comparável à taxa de processamento, usando memórias mais rápidas, memórias mais largas, compactação de dados e caches melhores. A combinação dessas quatro técnicas historicamente tem permitido que o desempenho da GPU dobre regularmente, a cada 12 a 18 meses mais ou menos. Essa taxa, ultrapassando a taxa da lei de Moore, foi demonstrada em aplicações gráficas por aproximadamente dez anos, e não mostra sinais de diminuição de velocidade significativa. O limitador de taxa mais desafiador parece ser o sistema de memória, mas a inovação competitiva também está avançando isso rapidamente.

Falácia: a GPUs exibem gráficos 3D; elas não podem fazer cálculos gerais. As GPUs são criadas para exibir gráficos 3D e também gráficos e vídeo 3D. Para atender as demandas do desenvolvedor de software gráfico expressas nas interfaces e requisitos de desempenho/recursos das APIs gráficas, as GPUs se tornaram processadores de ponto flutuante programáveis maciçamente paralelos. No domínio gráfico, esses processadores são programados por meio das APIs gráficas e com linguagens de programação gráfica misteriosas (GLSL, Cg e HLSL, em OpenGL e Direct3D). Porém, não há nada que impeça os arquitetos de GPU de exporem os cores de processador paralelo a programadores sem a API gráfica ou as linguagens gráficas misteriosas.

De fato, a família de GPUs da arquitetura Tesla expõe os processadores por meio de um ambiente de software conhecido como CUDA, que permite que os programadores desenvolvam programas de aplicação gerais usando a linguagem C e logo C++. As GPUs são processadores completos de Turing, de modo que podem executar qualquer programa que uma CPU pode executar, embora talvez não tão bem. E talvez mais rápido.

Falácia: GPUs não podem executar programas de ponto flutuante com precisão dupla com rapidez. No passado, as GPUs não podiam executar programa algum de ponto flutuante com precisão dupla. E isso não é muito rápido de forma alguma. As GPUs progrediram da representação aritmética indexada (tabelas de pesquisa para cores) para inteiros de 8 bits por componente de cor, aritmética de ponto fixo, ponto flutuante com precisão simples e, recentemente, adicionaram a precisão dupla. As GPUs modernas realizam praticamente todos os cálculos em aritmética de ponto flutuante IEEE com precisão simples, e estão começando a usar também a precisão dupla.

Por um pequeno custo adicional, uma GPU pode admitir ponto flutuante com precisão dupla além de ponto flutuante com precisão simples. Hoje, a precisão dupla roda mais lentamente que a velocidade de precisão simples, cerca de cinco a dez vezes mais lenta. Pelo custo adicional incremental, o desempenho da precisão dupla pode ser aumentado com relação à precisão simples em estágios, conforme a maioria das aplicações exige.

Falácia: GPUs não computam ponto flutuante corretamente. As GPUs, pelo menos na família de processadores da arquitetura Tesla, realizam o processamento de ponto flutuante com precisão simples em um nível prescrito pelo padrão de ponto flutuante IEEE 754. Assim, em termos de precisão, as GPUs são iguais a quaisquer outros processadores compatíveis com IEEE 754.

Hoje, as GPUs não implementam alguns dos recursos específicos descritos no padrão, como o tratamento de números desnormalizados e o oferecimento de exceções de

ponto flutuante precisas. Porém, a GPU Tesla T10P, introduzida recentemente, oferece arredondamento IEEE completo, multiplicação-adição fundida e suporte para número desnormalizado para precisão dupla.

Armadilha: só use mais threads para cobrir latências de memória maiores. Os cores da CPU normalmente são projetados para executar uma única thread em velocidade total. Para executar em velocidade total, cada instrução e seus dados precisam estar disponíveis quando for o momento para essa instrução executar. Se a próxima instrução não estiver pronta ou os dados exigidos para essa instrução não estiverem disponíveis, a instrução não poderá ser executada e o processador emite um stall. A memória externa está distante do processador, de modo que exige muitos ciclos de execução desperdiçados para apanhar dados da memória. Consequentemente, as CPUs exigem grandes caches locais para continuar executando sem stalling. A latência da memória é longa, de modo que é evitada esforçando-se para executar na cache. Em algum ponto, as demandas do conjunto de trabalho do programa podem ser maiores que qualquer cache. Algumas CPUs têm usado o multithreading para tolerar a latência, mas o número de threads por core geralmente tem sido limitado a um pequeno número.

A estratégia da GPU é que diferentes cores da GPU sejam projetados para executar muitas threads simultaneamente, mas apenas uma instrução de qualquer thread de uma vez. Outra forma de dizer isso é que uma GPU executa cada thread lentamente, mas em conjunto executa as threads de uma maneira mais eficiente. Cada thread pode tolerar alguma quantidade de latência da memória, pois outras threads podem executar.

A desvantagem disso é que múltiplas threads são necessárias para cobrir a latência da memória. Além disso, se os acessos à memória forem espalhados e não correlacionados entre as threads, o sistema de memória será cada vez mais lento na resposta a cada solicitação individual. Por fim, até mesmo as múltiplas threads não serão capazes de cobrir a latência. Assim, a armadilha é que, para a estratégia de trabalho “só usar mais threads” para cobrir a latência, você precisa ter threads suficientes, e as threads precisam ser bem comportadas em termos de localidade do acesso à memória.

Falácia: algoritmos $O(n)$ são difíceis de aumentar a velocidade. Não importa como a GPU é rápida no processamento de dados, as etapas de transferência de dados de e para o dispositivo podem limitar o desempenho dos algoritmos com a complexidade $O(n)$ (com uma pequena quantidade de trabalho por dado). A taxa de transferência mais alta em relação ao barramento PCIe é aproximadamente 48GB/segundo quando são usadas transferências de DMA, e ligeiramente menor para transferências não DMA. A CPU, pelo contrário, tem velocidades de acesso típicas de 8-12GB/segundo para a memória do sistema. Problemas de exemplo, como a adição em vetor, serão limitados pela transferência de entradas à GPU e a saída de retorno do cálculo.

Há três maneiras de contornar o custo de transferência de dados. Primeiro, tente deixar os dados na GPU pelo máximo de tempo possível, em vez de mover os dados para lá e para cá em diferentes etapas de um algoritmo complicado. CUDA deliberadamente deixa dados na GPU entre as partidas para dar suporte a isso.

Segundo, a GPU admite as operações simultâneas de copy-in, copy-out e cálculo, de modo que os dados podem ser enviados para dentro e fora do dispositivo enquanto ele está calculando. Esse modelo é útil para qualquer stream de dados que possa ser processado enquanto eles chegam. Alguns exemplos são processamento de vídeo, roteamento de rede, compactação/descompactação de dados e até mesmo cálculos simples, como matemática com grandes vetores.

A terceira sugestão é usar a CPU e a GPU juntas, melhorando o desempenho ao atribuir um subconjunto do trabalho a cada um, tratando o sistema como uma plataforma de computação heterogênea. O modelo de programação CUDA admite a alocação de trabalho a uma ou mais GPUs junto com o uso continuado da CPU, sem o uso de threads (por meio de funções assíncronas da GPU), de modo que é relativamente simples manter todas as GPUs e uma CPU trabalhando simultaneamente para solucionar problemas ainda mais rapidamente.

A.10 Comentários finais

As GPUs são processadores maciçamente paralelos e se tornaram bastante utilizados, não apenas para gráficos 3D, mas também para muitas outras aplicações. Essa ampla aplicação foi possibilitada pela evolução de dispositivos gráficos nos processadores programáveis. O modelo de programação de aplicação gráfica para GPUs normalmente é uma API como DirectX™ ou OpenGL™. Para uma computação de uso geral, o modelo de programação CUDA usa um estilo SPMD (Single-Program Multiple Data), executando um programa com muitas threads paralelas.

O paralelismo da GPU continuará a se expandir segundo a lei de Moore, principalmente aumentando o número de processadores. Somente os modelos de programação paralela que podem se expandir prontamente para centenas de cores de processador e milhares de threads terão sucesso no suporte a GPUs e CPUs manycore. Além disso, somente as aplicações que tiverem muitas tarefas paralelas em grande parte independentes serão aceleradas pelas arquiteturas manycore maciçamente paralelas.

Os modelos de programação paralela para GPUs estão se tornando mais flexíveis, tanto para gráficos quanto para a computação paralela. Por exemplo, CUDA está evoluindo rapidamente na direção da funcionalidade plena do C/C++. APIs gráficas e modelos de programação provavelmente se adaptarão às capacidades de computação paralela e modelos CUDA. Seu modelo de threading no estilo SPMD é escalável, e é um modelo conveniente, sucinto e facilmente aprendido, para expressar grandes quantidades de paralelismo.

Impulsionada por essas mudanças nos modelos de programação, a arquitetura de GPU, por sua vez, está se tornando mais flexível e mais programável. As unidades de função fixa da GPU estão se tornando acessíveis a partir de programas em geral, acompanhando o modo como os programas CUDA já utilizam funções intrínsecas de textura para realizar pesquisas de textura usando a instrução de textura e unidade de textura da GPU.

A arquitetura da GPU continuará a se adaptar aos padrões de uso de gráficos e outros programadores de aplicação. As GPUs continuarão a se expandir para incluir mais poder de processamento através de cores de processador adicionais, além de aumentar a largura de banda de thread e memória disponível para os programas. Além disso, os modelos de programação precisam evoluir para incluir a programação de sistemas manycore heterogêneos, incluindo GPUs e CPUs.

Agradecimentos

Este apêndice é o trabalho de vários autores sobre NVIDIA. Somos gratos às contribuições significativas de Michael Garland, John Montrym, Doug Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman e Vasily Volkov.

A.11

Perspectiva histórica e leitura adicional

Esta seção, que aparece no site, analisa a história das unidades de processamento gráfico (GPUs) em tempo real programáveis desde o início dos anos 80 até hoje, à medida que diminuíram de preço por duas ordens de grandeza e aumentaram o desempenho também por duas ordens de grandeza. Ela acompanha a evolução da GPU desde pipelines de função fixa até os processadores gráficos programáveis, com perspectivas sobre computação de GPU, gráficos unificados e processadores de cálculo, computação visual e GPUs escaláveis.