5
Unidades de Processamento Gráfico – GPUs

As GPUs são processadores maciçamente paralelos, com múltiplos elementos de processamento, tipicamente utilizadas como aceleradores de computação. Elas fornecem um elevado poder computacional e têm sido usadas com sucesso em aplicações paralelas de propósito geral em diversas áreas. Embora diferentes fabricantes tenham desenvolvido GPUs nos últimos anos, optamos pelas GPUs da nVidia devido à sua flexibilidade e disponibilidade.

A Figura 5.1 apresenta uma comparação de desempenho entre CPU e GPU em bilhões de operações de ponto flutuante por segundo.

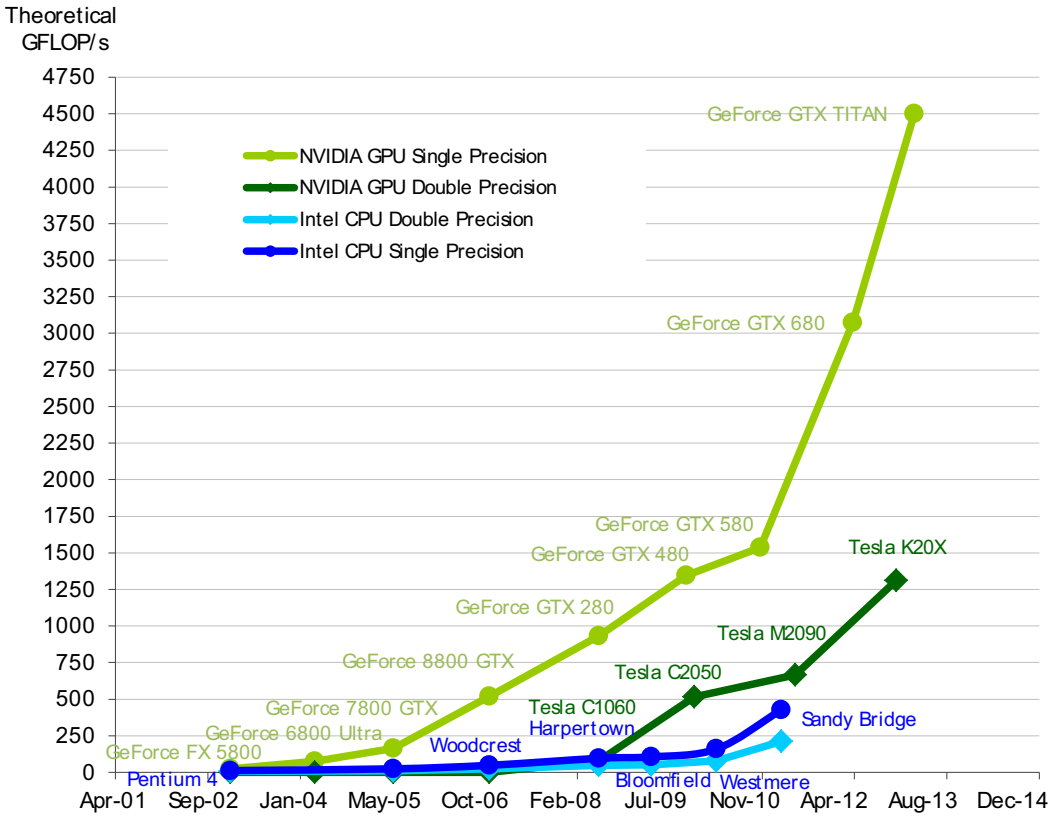

Figura 5.1: Comparação de desempenho entre CPU e GPU em bilhões de operações de ponto flutuante por segundo. Fonte: nVidia [80].

A GPU GeForce GTX TITAN apresenta um desempenho em precisão simples de ponto flutuante da ordem de 4,5 TFlops, muito superior aos cerca de 500 GFlops apresentados pela CPU Sandy Bridge. O desenvolvimento da GPU foi impulsionado pelo mercado dos jogos para computador que demanda imagens de alta definição em tempo real. Assim, a GPU se tornou um

hardware especializado para realizar tarefas altamente paralelas, destinando mais transistores para processamento, enquanto que a CPU possui uma área muito maior destinada a controle de fluxo e cache de dados.

A Figura 5.2 apresenta uma comparação entre a largura de banda da CPU e da GPU em bilhões de bytes por segundo.

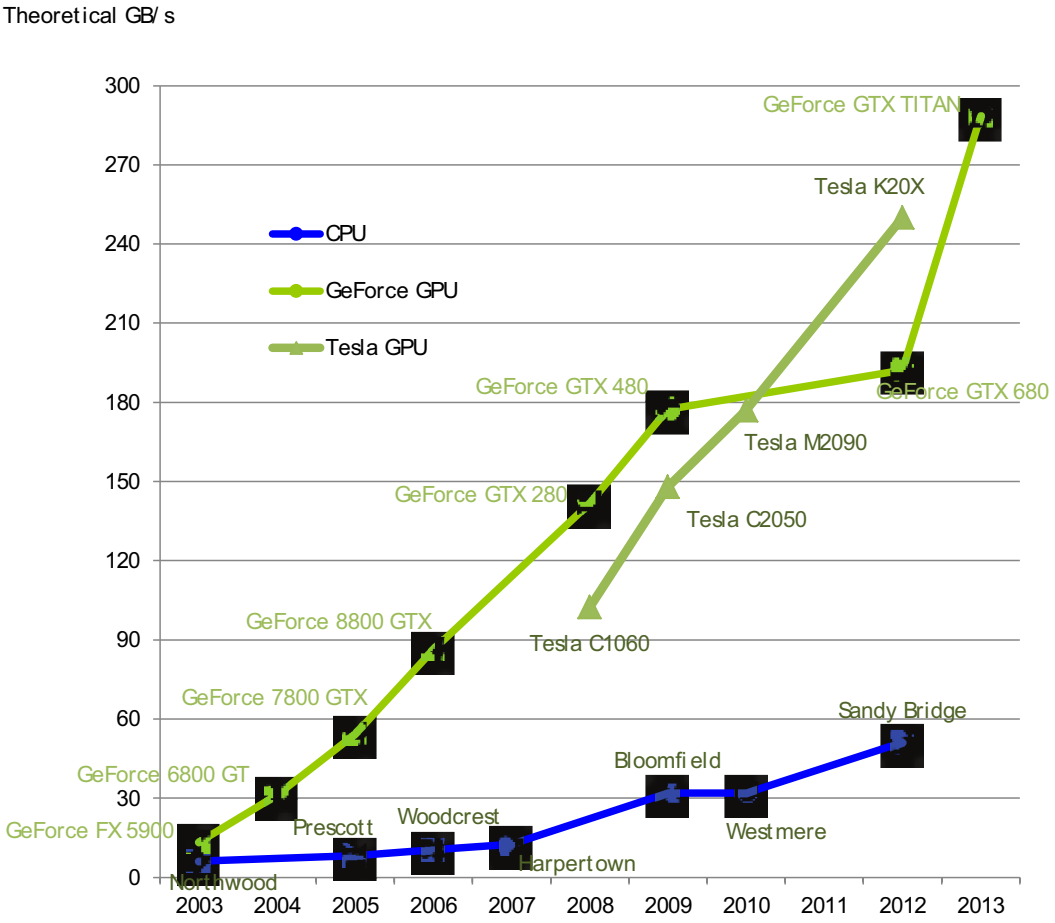

Figura 5.2: Comparação da largura de banda da memória entre CPU e GPU em bilhões de bytes por segundo. Fonte: nVidia [80].

GPU GeForce GTX TITAN apresenta uma velocidade de acesso a memória de 288,4 GB/s, muito superior aos cerca de 50 GB/s apresentados pela CPU Sandy Bridge. Essa maior velocidade de acesso permite que a GPU seja capaz de manter os seus elementos de processamento ocupadas com dados a serem processados. O controle de fluxo e a cache da GPU é menos sofisticado quando comparado à uma CPU, por isso a GPU apresenta bom desempenho para problemas que podem ser expressos como paralelismo de dados escondendo a latência de acesso à memória com a realização maciça de operações de ponto flutuante.

5.1 Arquitetura da GPU

As GPUs da nVidia são tipicamente compostas por um conjunto de *streaming multiprocessors* (SMs), cada um sendo composto por um conjunto de elementos de processamento (*scalar processors*–SPs). Os SPs podem ser vistos como núcleos de processamento relativamente simples quando comparados aos complexos núcleos das CPUs.

A arquitetura de uma GPU Kepler com o chip GK110 é composta por 15 SMs e 6 controladores de memória de 64 bits [81]. Algumas variações do chip GK110 podem possuir 13 ou 14 SMs. A Figura 5.3 ilustra um SM da arquitetura de GPUs Kepler. Cada SM é composto por 192 SPs, capazes de realizar operações com precisão simples de ponto flutuante, lógica e aritmética de inteiros. Há 64 unidades para realizar instruções em precisão dupla de ponto flutuante, 32 unidades para funções especiais capazes de realizar instruções complexas e 32 unidades de leitura e escrita.

A Figura 5.4 apresenta a organização da memória da GPU, podendo ser descrita da seguinte forma: uma grande memória global com elevada latência; uma memória muito rápida, de baixa latência, integrada ao chip chamada de memória compartilhada para cada SM; e uma memória local privada para cada *thread*. Além disso, uma GPU Kepler com o chip GK110 possui alguns espaços adicionais de memória para implementar caches, tais como um espaço de memória com 64 KB que pode ser configurado entre a memória cache L1 e a memória compartilhada. As possíveis configurações são: 32 KB para cada uma das duas; 48 KB de cache L1 com 16 KB de memória compartilhada; ou 16 KB de cache L1 com 48 KB de memória compartilhada. Uma cache L2 com 1536 KB através da qual são realizadas as leituras e escritas dos dados. E uma memória cache de 48 KB, somente leitura, usada para implementar ambas as caches de textura e de constantes.

Toda GPU deve estar acoplada a uma CPU. A GPU necessita do controle da CPU para iniciar seu processamento. O termo *host* é comumente empregado para designar a CPU na qual a GPU está instalada. A transferência de dados entre a CPU e a GPU é realizada através do barramento PCIe. A CPU e a GPU possuem espaços de memória RAM separados, chamados de memória do *host* e memória do dispositivo, e o tempo de transferência entre CPU-GPU é limitado pela velocidade do barramento PCIe.

Figura 5.3: SM (*streaming multiprocessor*) da arquitetura de uma GPU Kepler com o chip GK110. Fonte: nVidia [81].

5.2 Modelo de Programação

O modelo de programação da nVidia é chamado de CUDA (*Computer Unified Device Architecture*) [80]. CUDA é um ambiente de desenvolvimento baseado em C que permite que o programador defina funções especiais em C, chamadas de *kernels*, as quais executam em paralelo na GPU utilizando diferentes *threads*. A GPU suporta um grande número de *threads* com paralelismo finamente granulado.

As *threads* são organizadas em uma hierarquia de grupos de *threads*, conforme ilustrado pela Figura 5.5. *Threads* são divididas em uma *grid* bi ou tridimensional de **blocos de threads**. Cada bloco de *threads* é um conjunto bi ou tridimensional de *threads*. Os blocos de *threads* são executados

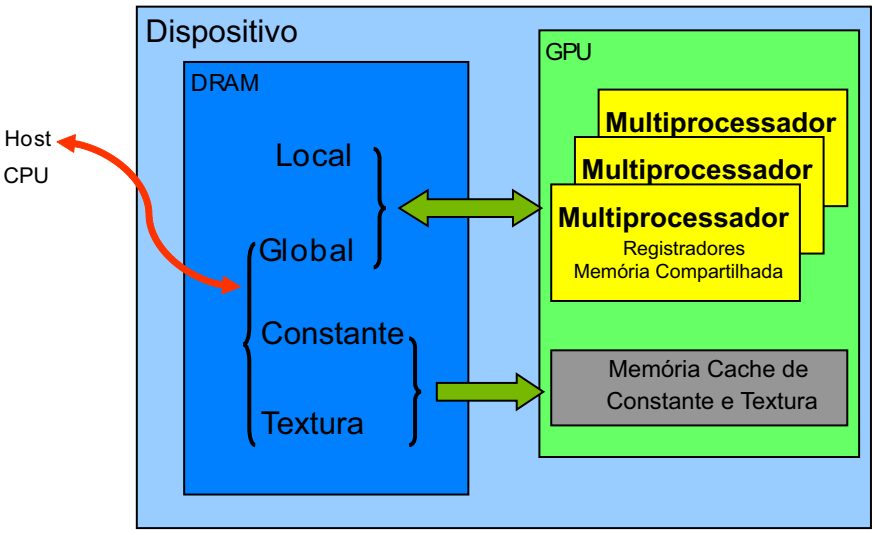

Figura 5.4: Organização dos espaços de memória de uma GPU. Adaptado de: nVidia [82].

na GPU através da atribuição de um determinado número de blocos para serem executados num SM. Cada *thread* de um bloco de *threads* tem um identificador único, dado por variáveis internas `threadIdx.x, threadIdx.y` e `threadIdx.z`. Cada bloco de *threads* tem um identificador único que distingue sua posição na *grid*, dado por variáveis internas `blockIdx.x, blockIdx.y` e `blockIdx.z`. As dimensões da *grid* e do bloco de *threads* são especificadas no momento em que o *kernel* é carregado através dos identificadores `gridDim` e `blockDim`, respectivamente.

Todas as *threads* em um bloco de *threads* são atribuídas para serem executas em um mesmo SM. Por este motivo, as *threads* dentro de um bloco podem cooperar entre elas através da utilização de primitivas de sincronização e através do uso da memória compartilhada. Entretanto, o número de *threads* dentro de um bloco pode exceder o número de elementos de processamento em um SM, sendo necessário um mecanismo de *scheduling*. Este mecanismo de *scheduling* divide o bloco em *warps*.

Cada *warp* tem um número fixo de *threads* agrupadas por identificadores consecutivos. O *warp* é executado em um SM de uma maneira SIMD implícita, chamada de SIMT (*Single Instruction Multiple Threads*). Todos os elementos de processamento do SM executam a mesma instrução simultaneamente, porém sobre elementos de dados diferentes. Entretanto, as *threads* podem logicamente seguir diferentes caminhos de controle de fluxo e são livres para divergir. Se alguma das *threads* executando em paralelo dentro do *warp* escolher um caminho de execução diferente, chamado de **divergência de**

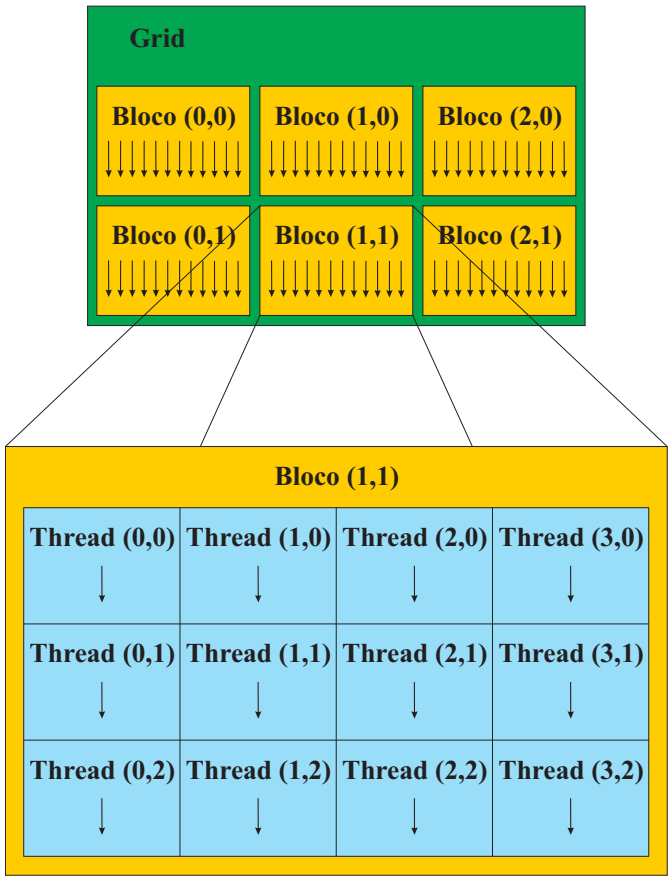

Figura 5.5: Hierarquia de organização das *threads*: grid e blocos de *threads*. Adaptado de: nVidia [80].

código, a execução das *threads* é serializada. Neste caso, o *warp* precisa ser carregado múltiplas vezes, uma para cada grupo de *threads* divergentes. Então, a eficiência completa só é conseguida quando todas as *threads* do *warp* seguem o mesmo caminho de execução. Caso contrário, a eficiência do paralelismo é significativamente reduzida.

Na arquitetura de uma GPU Kepler com o chip GK110, cada SM possui 4 escalonadores de *warps* (*warp schedulers*) e 8 unidades de despacho de instruções [81]. Isto significa que cada SM pode selecionar e executar simultaneamente 4 *warps* de 32 *threads* por vez. Além disso, é possível executar simultaneamente duas instruções por *thread* do *warp*, desde que elas sejam independentes e utilizem unidades diferentes do hardware. Por exemplo, cada *thread* de um *warp* pode executar simultaneamente uma instrução de precisão simples e a outra de precisão dupla de ponto flutuante. Existe também a possibilidade de se executar simultaneamente duas instruções independentes, se uma delas utilizar a unidade lógica e aritmética de inteiros, enquanto a outra utilizar a unidade de ponto flutuante.

5.3 Compilação

A compilação de um programa CUDA é realizada através dos seguintes estágios. Primeiro, a interface de CUDA, chamada de *cudafe*, divide o programa em duas partes, uma parte composta por código C/C++ a ser executado no *host* e a outra parte composta por código para o dispositivo a ser executado na GPU. O código para o *host* é compilado com um compilador de C comum, tal como o *gcc*. Conforme ilustrado na Figura 5.6, o código para o dispositivo é compilado usando o compilador CUDA, chamado de *nvcc*, gerando um código intermediário numa linguagem pseudo-assembly chamada de PTX (*Parallel Thread Execution*). PTX é uma linguagem de programação de baixo nível semelhante ao assembly, usada para GPUs da nVidia e que precisa ser compilada, escondendo muitos detalhes do hardware. Esta linguagem PTX apresenta boa documentação disponibilizada pela nVidia. Numa última etapa, o código PTX é compilado para o código binário da GPU, chamado de CUBIN, através do uso do compilador *ptxas*.

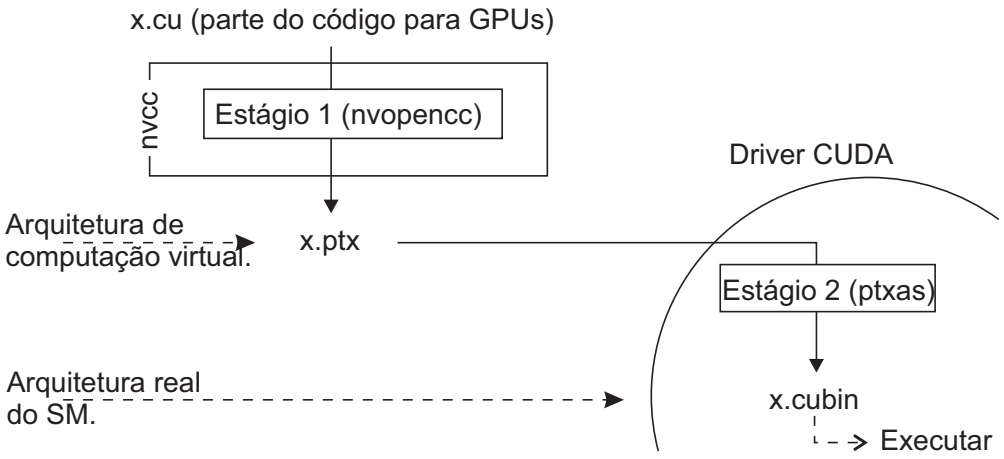

Figura 5.6: Etapas de compilação da parte do código destinada a executar na GPU.

Ao contrário da linguagem PTX, cuja documentação é pública, o formato CUBIN é proprietário. A nVidia não disponibiliza informações sobre CUBIN. Além disso, o fabricante fornece apenas os elementos mais básicos da arquitetura do hardware e aparentemente não há planos de disponibilizar mais informações no futuro.