Arquitecturas de

Programação CUDA

António Rui Borges

Resumo

- *CUDA*
- *Modelo de programação CUDA*
- *Leitura sugerida*

CUDA - 3

CUDA é uma plataforma de computação paralela de uso geral e um modelo de programação que aproveita o motor do computador paralelo nas GPUs NVIDIA para resolver muitos problemas de computação intensiva de uma forma mais eficiente.

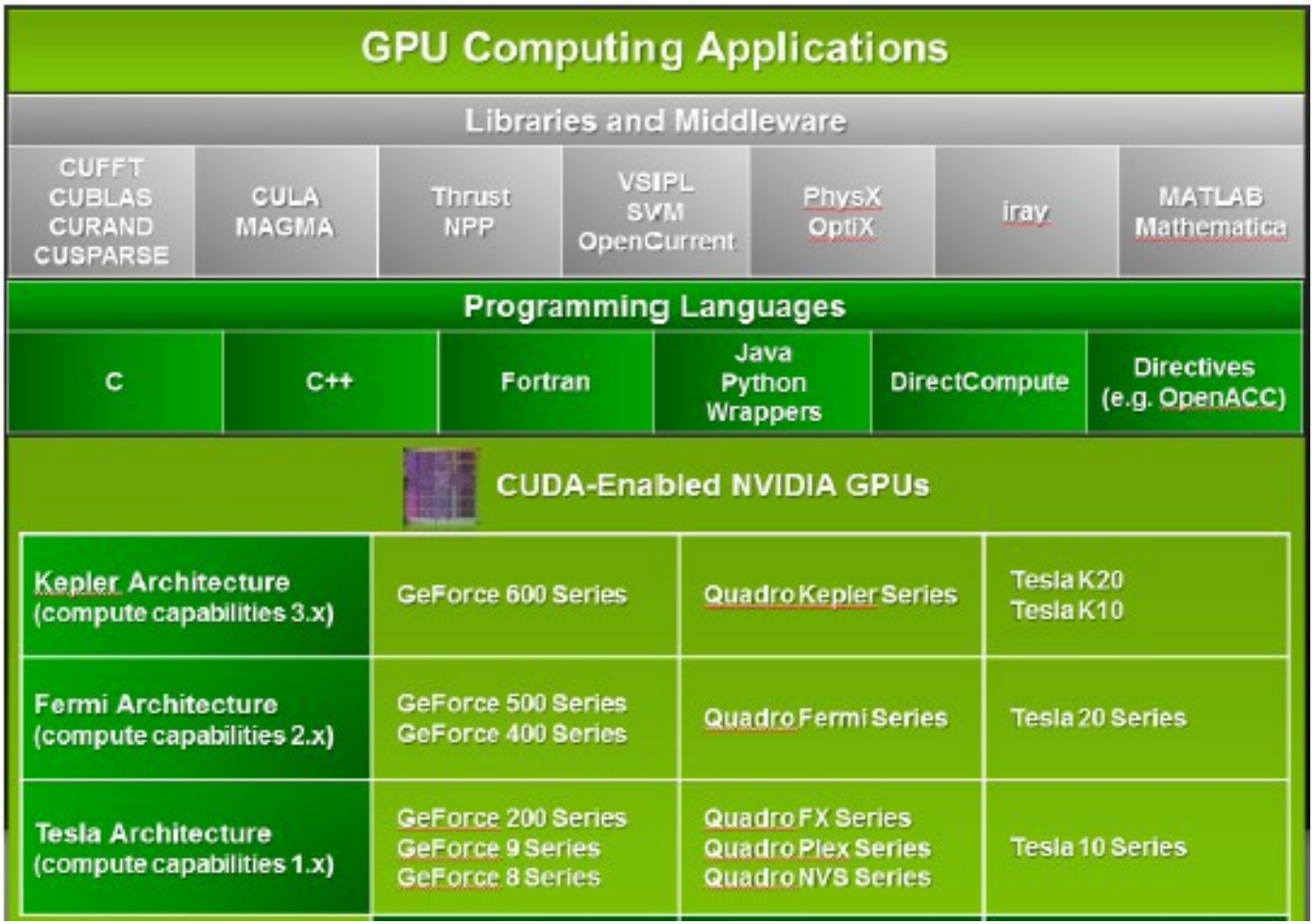
A plataforma CUDA é acessível através de bibliotecas aceleradas por CUDA, directivas de compilação, interfaces de programação de aplicações e extensões a linguagens de programação padrão da indústria como C, C++, Fortran, Java e Python.

CUDA C é uma extensão da norma ANSI C com um punhado de extensões de linguagem para permitir uma programação heterogénea e também APIs simples para gerir dispositivos, memória e outras tarefas.

CUDA - 4

Plataforma CUDA para computação heterogénea

Fonte: Guia de Programação CUDA C - Nvidia



CUDA - 5

A CUDA fornece dois níveis API para gerir o dispositivo GPU e organizar os fios

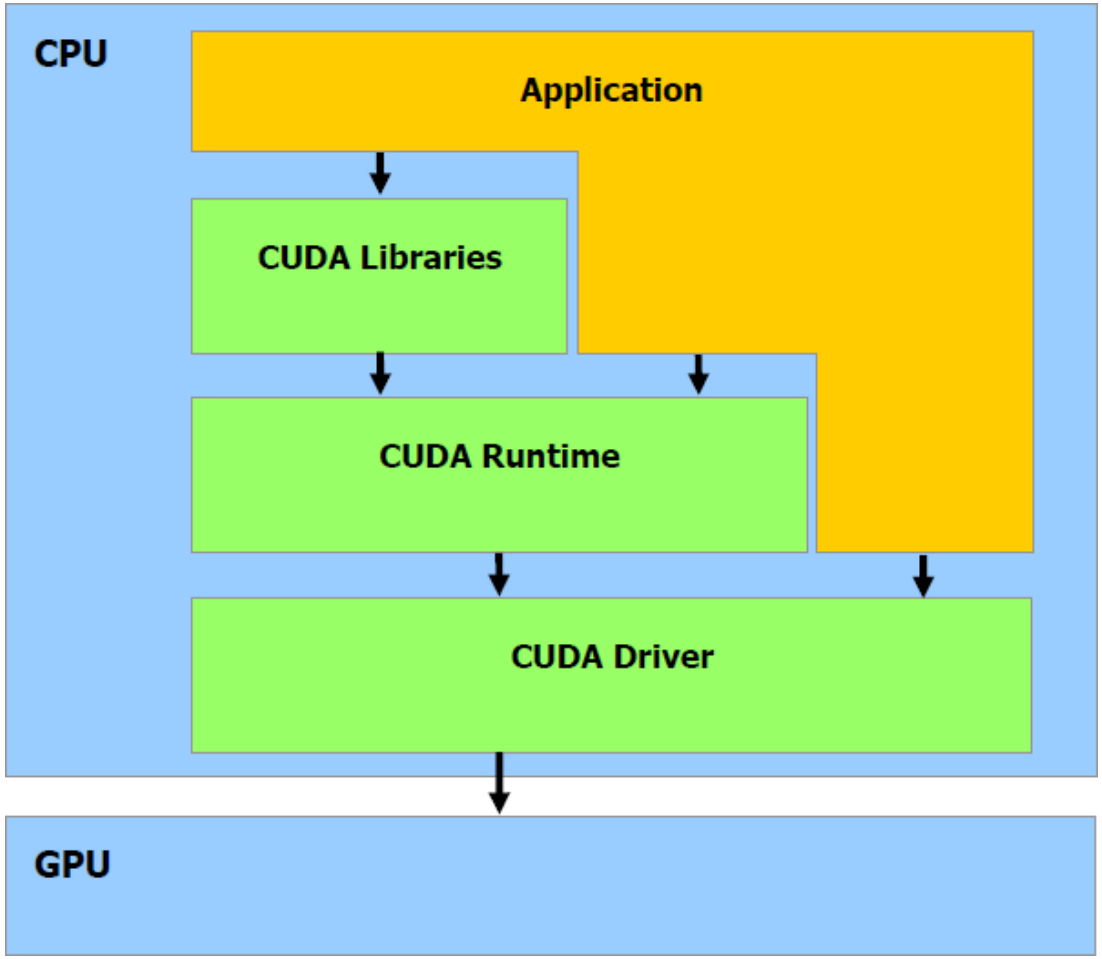
- CUDA driver API (http://docs.nvidia.com/cuda/cuda-driver-api/index.html)
- CUDA runtime API (http://docs.nvidia.com/cuda/cuda-runtime-api/index.html).

O *driver* API é um API de baixo nível e é relativamente difícil de programar. Contudo, dá ao programador mais controlo sobre a forma como o dispositivo GPU é utilizado. O API de *tempo de execução*, por outro lado, é um API de alto nível implementado em cima do API do driver. Cada função do API de tempo de execução é tipicamente dividida em uma ou mais operações básicas emitidas para o API do controlador.

Não há nenhuma diferença perceptível entre eles. A forma como os programadores organizam os fios e como a memória é utilizada tem um efeito muito mais pronunciado no perfor- mance. No entanto, são mutuamente exclusivas: apenas uma delas pode ser usada no código fonte, não é permitida nenhuma mistura de chamadas de função de ambas.

Interface de software CUDA

Fonte: Programação profissional CUDA C



CUDA - 7

O compilador CUDA `nvcc` da NVIDIA separa o código do dispositivo do código do anfitrião durante o processo de compilação. O *código do hospedeiro* é o código C padrão e é compilado posteriormente com um compilador C. O *código do dispositivo* é escrito usando CUDA C alargado com palavras-chave para etiquetar funções paralelas de dados, chamadas *kernels*. O código do dispositivo é ainda compilado pelo `nvcc`. Durante a fase de ligação, são adicionadas bibliotecas CUDA de tempo de execução ou drivers para chamadas de procedimento de kernel e manipulação explícita da GPU.

O conjunto de ferramentas CUDA inclui o compilador, bibliotecas matemáticas e ferramentas para a depuração e optimização do desempenho das aplicações.

CUDA - 8

No seu núcleo, a CUDA tem três abstracções chave: uma hierarquia de grupos de fios, memórias partilhadas, e sincronização de barreiras - que são simplesmente expostas ao programador como um conjunto mínimo de extensões de linguagem.

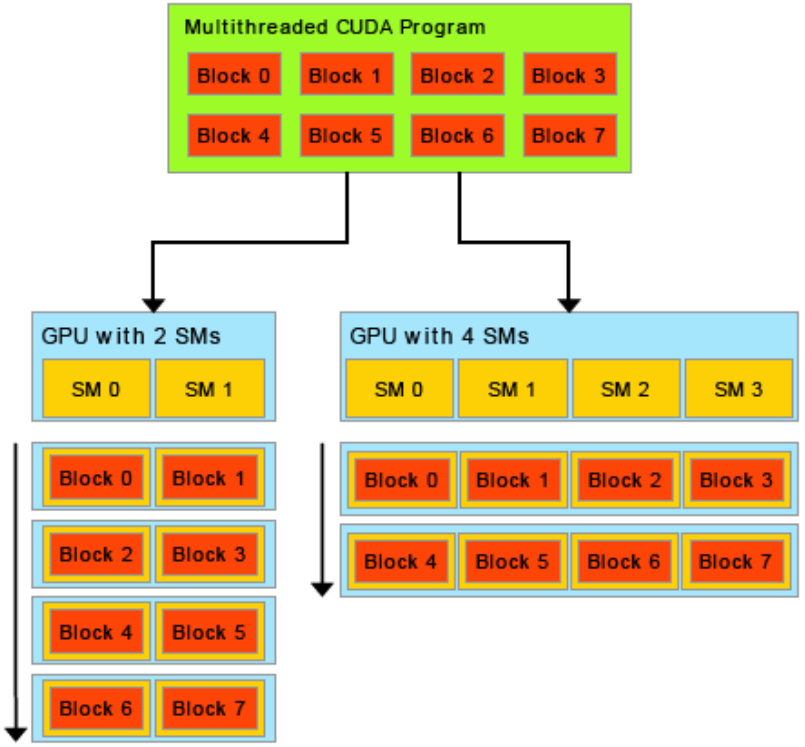
As abstracções acima mencionadas fornecem paralelismo de dados de grão fino e paralelismo de fios, aninhados dentro do paralelismo de dados de grão grosseiro e paralelismo de tarefas. Permitem ao programador dividir o problema num conjunto de sub-problemas, que podem ser resolvidos independentemente em paralelo por blocos de fios, e cada sub-problema por sua vez em peças ainda mais finas que podem ser resolvidas de forma cooperativa em paralelo por todos os fios dentro do bloco.

Esta decomposição preserva a expressividade da linguagem, permitindo que os fios cooperem na resolução de cada subproblema e, ao mesmo tempo, permite a disponibilidade automática. De facto, cada bloco de fios pode ser programado em qualquer um dos multiprocessadores disponíveis dentro de uma GPU em qualquer ordem, concomitantemente ou sequencialmente, de modo a que um programa CUDA compilado possa executar em qualquer número de multiprocessadores e apenas o

sistema de tempo de execução precise de conhecer a contagem física de multiprocessadores.

CUDA - 10

Uma GPU é construída em torno de um conjunto de *multiprocessadores de streaming*, organizados como uma topologia MIMD de processadores SIMD. Um programa multithreaded é dividido em blocos de fios que executam independentemente uns dos outros, de modo que uma GPU com mais multiprocessadores executará automaticamente o programa em menos tempo do que uma GPU com menos multiprocessadores.

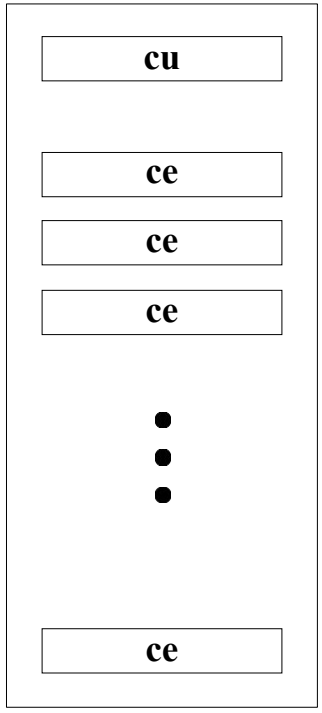


Escalabilidade automática

Fonte: Guia de Programação CUDA C - Nvidia

CUDA - 11

Um *multiprocessador de streaming* é essencialmente um processador SIMD: a mesma instrução será executada em paralelo em múltiplos *elementos informáticos* ou *núcleos*, cada um tendo o seu próprio banco de registo e unidades funcionais. Neste contexto, um *fio* pode ser pensado como as operações realizadas em sequência em cada elemento informático.



Cada *bloco de fios* é executado no mesmo multiprocessador de streaming e é dividido em blocos de 32 fios, chamados *urdiduras*, que são processados em paralelo.

CUDA - 12

Uma estrutura típica do programa CUDA consiste em cinco passos principais

- alocar memória GPU
- copiar dados da memória da CPU para a memória GPU
- invocar o núcleo CUDA para realizar um cálculo específico do programa
- copiar os dados de volta da memória GPU para a memória da CPU
- destruir as memórias da GPU.

Olá mundo

```
#incluir ".../comum/comum.h"
#incluir <stdio.h>
 */DETI
/* Uma simples introdução à programação na CUDA. Este programa
 * imprime "Hello World from GPU! from CUDA10 threads running
 * no GPU.


__olá globalvoidFromGPU()
{
    printf("Hello World from GPU!\n");
}

int main(int argc, char **argv)
{
    printf("Hello World from CPU!\n");

    oláFromGPU<<<1, 10>>>>();
    CHECK(cudaDeviceReset());
    retornar 0;
}
```

Modelo de programação CUDA - 14

CUDA C estende C permitindo ao programador definir funções C, chamadas *kernels*, que, quando chamadas, são executadas N vezes ao mesmo tempo por N fios CUDA diferentes, em oposição a apenas uma vez como funções C normais.

Um núcleo é definido usando `__global__` especificador da declaração. Ao telefonar, o número de fios CUDA que serão instanciados é especificado usando um `<<<...>>>` cláusula de configuração de execução que descreve uma grelha de blocos de rosca em execução concomitante.

A cada fio que executa o núcleo é dada uma identificação de fio única dentro do bloco de fios a que pertence. O ID da linha é acessível dentro do núcleo através da variável `threadIdx` incorporada. Cada thread dentro de um bloco de thread executa uma instância do kernel, tem os seus próprios registos e memória privada, dados de entrada e resultados de saída.

Modelo de programação CUDA - 15

Por conveniência, `threadIdx` é um vector de 3 componentes, para que os fios possam ser identificados utilizando um índice de fios unidimensional, bidimensional ou tridimensional, formando um bloco de fios unidimensional, bidimensional ou tridimensional, denominado *bloco de fios*. Isto proporciona uma forma natural de invocar um cálculo através dos elementos de um domínio como um vector, matriz, ou volume.

O índice de um fio e a sua identificação de fio relacionam-se de forma simples. Para um bloco unidimensional, eles são os mesmos; para um bloco bidimensional de tamanho (Dx, Dy), o ID de um fio de índice (x, y) é (x + y Dx); para um bloco tridimensional de tamanho (Dx, Dy, Dz), o ID de um fio de índice (x, y, z) é (x + y Dx + z Dx Dy).

Existe um limite para o número de fios por bloco, uma vez que todos os fios de um bloco devem residir no mesmo núcleo processador e devem partilhar os limitados recursos de memória desse núcleo. Nas GPUs actuais, um bloco de roscas pode conter até 1024 roscas. Contudo, um núcleo pode ser executado por múltiplos blocos de roscas com a mesma forma, de modo a que o número total de roscas seja igual ao número de roscas por bloco vezes o número de blocos.

Modelo de programação CUDA - 16

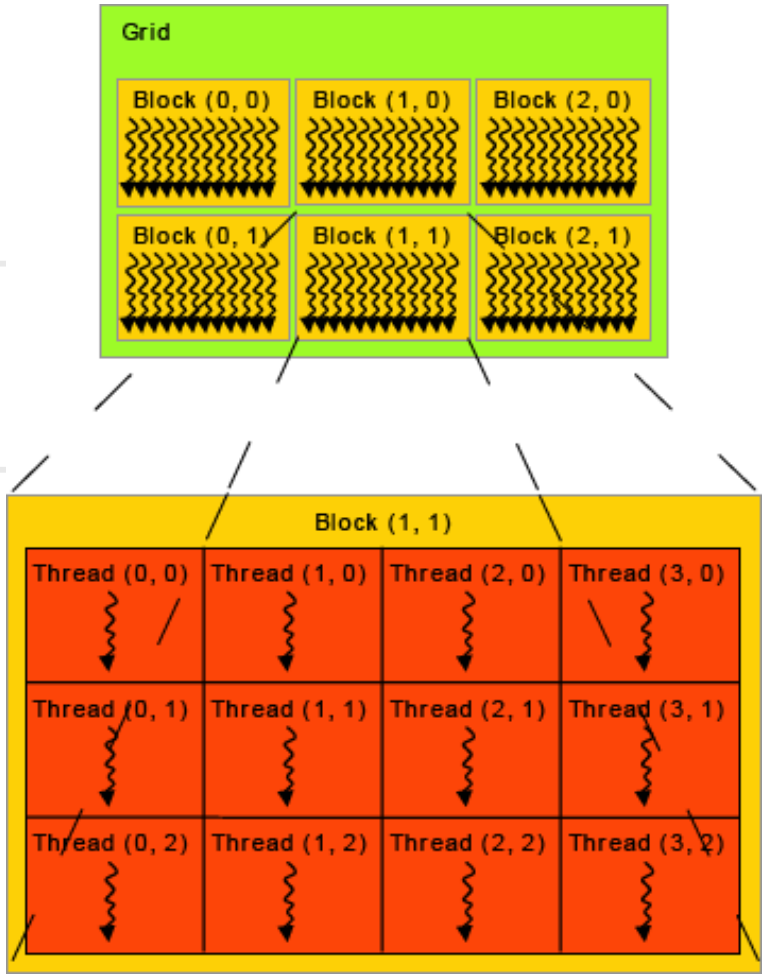
Os blocos estão organizados numa *grelha* 1-dimensional, 2-dimensional ou 3-dimensional de blocos de fios. O número de blocos de fios numa grelha é normalmente ditado pelo tamanho dos dados a processar ou o número de processadores no sistema, que pode exceder em muito.

O número de fios por bloco e o número de blocos por grelha especificado em `<<<...>>>` a sintaxe pode ser do tipo **`int`** ou `dim3`. Cada bloco dentro da grelha pode ser identificado por um índice unidimensional, bidimensional, ou tridimensional acessível dentro do núcleo através da variável `blockIdx` incorporada. A dimensão do bloco de rosca é acessível dentro do kernel através da variável `blockDim` vari- capaz.

Modelo de programação CUDA - 17

Grade de blocos de fios

Fonte: Guia de Programação CUDA C - Nvidia



Modelo de programação CUDA - 18

Os blocos de rosca são necessários para executar independentemente. Deve ser possível executá-los em qualquer ordem, em paralelo ou em série. Esta independência requer- ment permite que os blocos de fios sejam programados aleatoriamente em qualquer número de núcleos, permitindo aos programadores escrever códigos que escalonam com o número de núcleos.

As roscas dentro de um bloco podem cooperar partilhando dados através de alguma memória partilhada e sincronizando a sua execução para coordenar os acessos à memória. Mais precisamente, é possível especificar pontos de sincronização no kernel, chamando
estas síncopes `()` função intrínseca;
`__syncthreads()` actua como uma barreira
em que todos os fios do bloco devem esperar antes que qualquer um seja permitido prosseguir.

Para uma cooperação eficiente, espera-se que a memória partilhada seja uma memória de baixa latência perto de cada núcleo do processador (muito semelhante a uma cache L2 numa CPU convencional) e espera-se que `as`

Modelo de programação CUDA - 19

`sincronias()` sejam leves.

Modelo de programação CUDA - 20

Os fios CUDA podem aceder a dados de múltiplos espaços de memória durante a sua execução. Cada linha tem memória local privada. Cada bloco de fios tem memória partilhada visível a todos os fios do bloco e com a mesma vida útil que o bloco.

Todos os fios têm acesso à mesma *memória global*. Há também dois espaços adicionais de memória só de leitura acessíveis por todos os fios: os espaços de memória *constante* e de *textura.* Os espaços de memória global, constante, e de textura são optimizados para diferentes utilizações de memória.

A memória de textura também oferece diferentes modos de endereçamento, bem como filtragem de dados, para alguns formatos de dados específicos. Os espaços de memória global, constante e de textura são persistentes nos lançamentos do kernel através da mesma aplicação.

O modelo de programação CUDA também assume que tanto o anfitrião como o dispositivo mantêm os seus próprios espaços de memória separados em DRAM, referidos como *memória do anfitrião* e *memória do dispositivo*, respectivamente. Portanto, um programa gere os espaços de memória globais, de cons- tant, e de textura visíveis para os núcleos através de chamadas para o tempo de execução

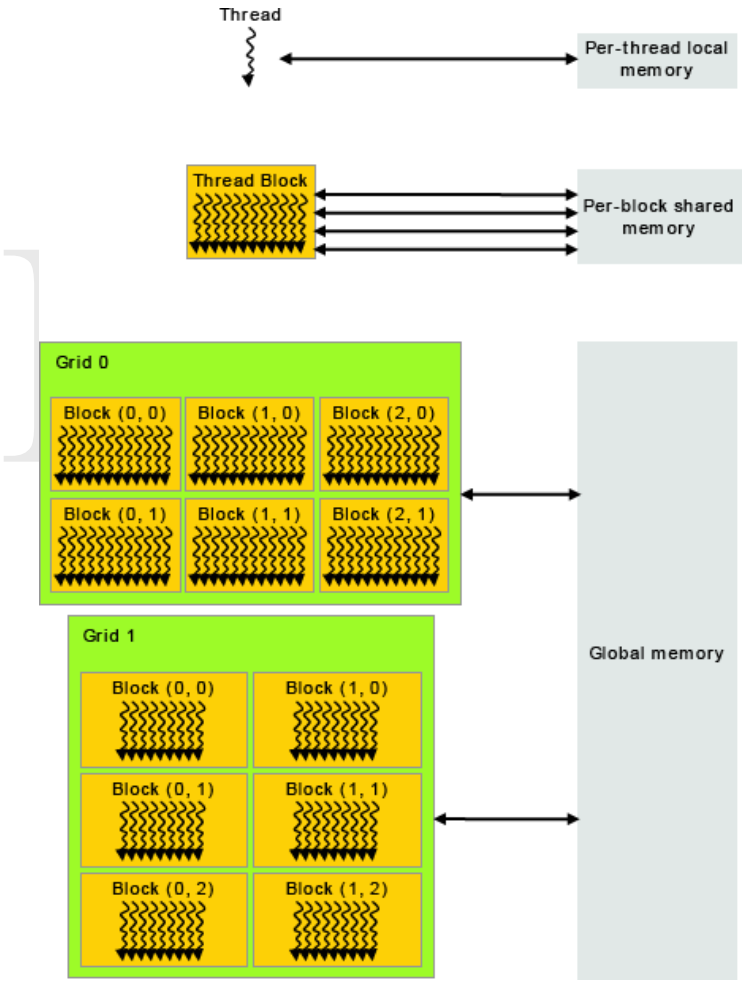
Modelo de programação CUDA - 21

CUDA. Isto inclui a alocação e desalocação da memória do dispositivo, bem como a transferência de dados entre a memória do anfitrião e a memória do dispositivo.

Modelo de programação CUDA - 22

Hierarquia da memória

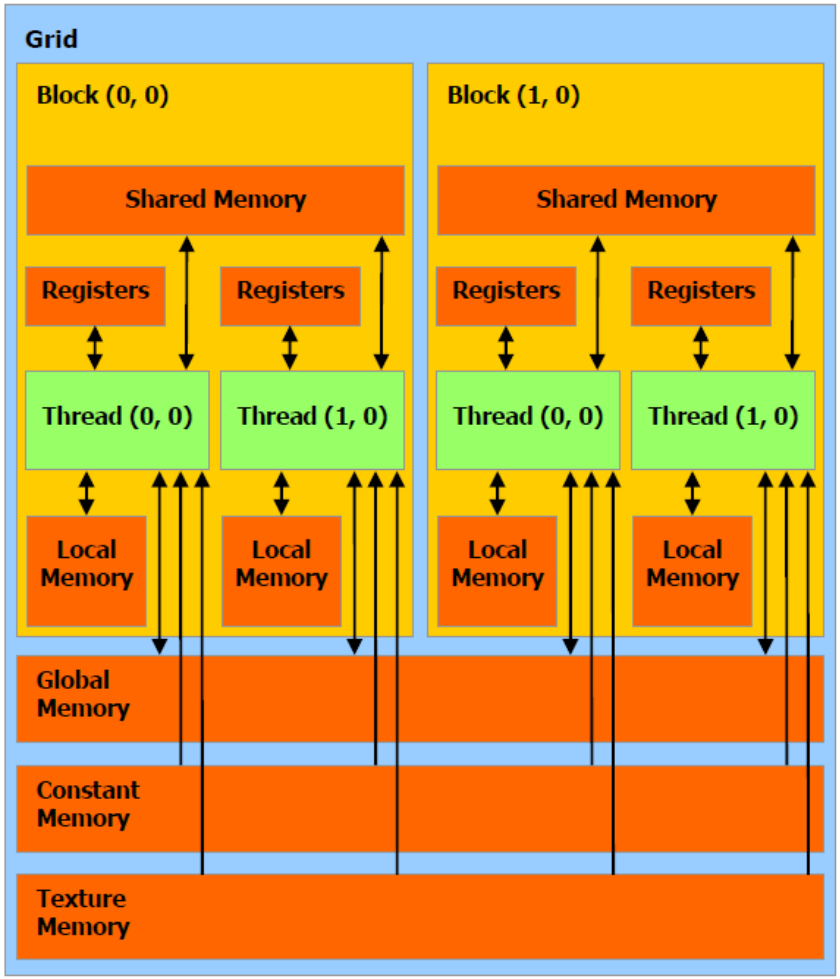
Fonte: Guia de Programação CUDA C - Nvidia



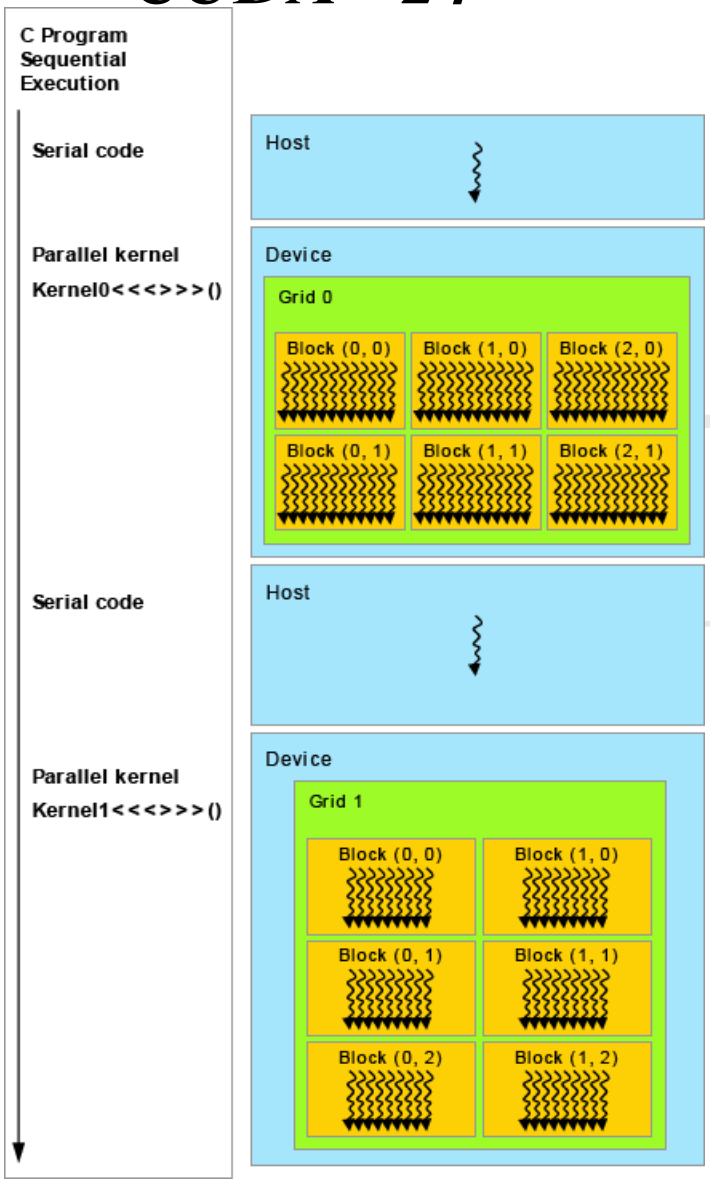
Modelo de programação CUDA - 23

Hierarquia da memória

Fonte: Guia de Programação CUDA C - Nvidia



Modelo de programação CUDA - 24



CPU - interacção GPU

Fonte: Guia de Programação CUDA C - Nvidia

Modelo de programação CUDA - 25

Qualificadores do tipo de função

Qualificador	Execução	Chamável	Observação
`__global__`	no dispositivo	do anfitrião	tipo de retorno **nulo**
`__device__`	no dispositivo	a partir do dispositivo	
`__host__`	sobre o anfitrião	do anfitrião	pode ser omitido

O `__device__` e a `__host__` os qualificadores podem ser utilizados em conjunto, caso em que a função é compilada tanto para o hospedeiro como para o dispositivo.

Modelo de programação CUDA - 26

Os grãos de CUDA são funções com restrições.

As seguintes restrições aplicam-se a todos os caroços

- apenas acedem à memória do dispositivo
- devem ter um tipo de retorno **nulo**
- não há suporte para um número variável de parâmetros
- não há suporte para variáveis estáticas
- não há apoio para apontadores de funções
- exibem um comportamento assíncrono.

Leitura sugerida

- *Guia de Programação CUDA C++*, NVIDIA, 2021
- *Programação Profissional CUDA*, Cheng J., Grossman M., McKercher T., John Wiley & Sons, Inc, 2014
- *Programação de Processadores Massivamente Paralelos: Uma abordagem prática*, Kirk D. B.., Hwu W. W., Morgan Kaufmann, 2017