High Performance Programming: Multicore, Clusters and GPU

GPGPU - CUDA

Professor Marcelo Trindade Rebonatto
Curso de Ciência da Computação

UPF

Sumário

- GPGPU
  - Introdução
  - CPU X GPU
  - Aplicações
- Como usar
- CUDA
  - Principais funções da API
- Exemplos

Introdução

- Placas gráficas
  - Jogos 3D evoluíram e passaram a exigir muito poder computacional
- Jogos
  - Além de gerar cenário 3D
  - Aplicação de texturas, iluminação, sombras, reflexões, ...
    - # Folhas individuais são desenhadas
    - # Sombras calculadas dinamicamente

Jogos ...

GPGPU

- O que é
  - General *Purpose Computing on GPUs*
  - Uso de hardware gráfico para computação não-gráfica
- GPU (*Graphics Processing Unit*)
  - Tradicionalmente usada para renderização em tempo real
  - Excelente atrativo para a exploração da programação paralela de aplicações

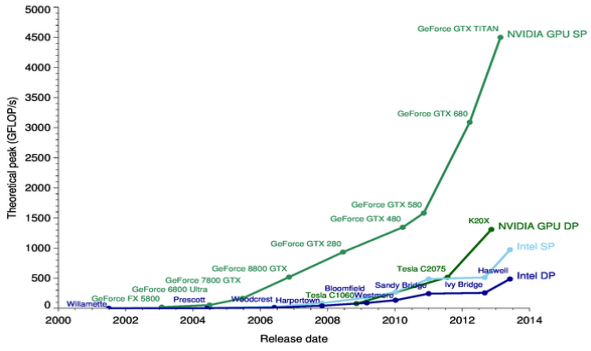
CPU x GPU

Modelo	Core 2 Quad 3 Ghz	GeForce 8800
Ano	2007	2006
Núcleos	4	128
Desempenho Aritmético	96 GFlops	330 GFlops

CPU x GPU

	Intel Core I7 980X	NVIDIA GTX 480
Núcleos	6	480
Desempenho aritmético (Máximo teórico)	0.1 Tflop (double)	0.5 Tflop (double) 1.3 Tflop (single)
Acesso Memória	25.6 GB/s	177 GB/s

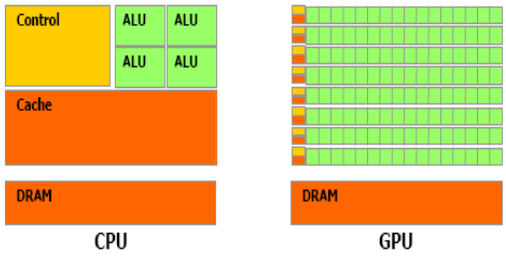
CPU x GPU?



CPU X GPU

- CPU
  - Quantidade de Caches (L1, L2, L3)
  - Previsão de salto
  - Alto desempenho (pelas previsões)
- GPU
  - Muitas ALUs
  - Memória OnBoard Rápida
  - Grande quantidade de tarefas paralelas

CPU x GPU



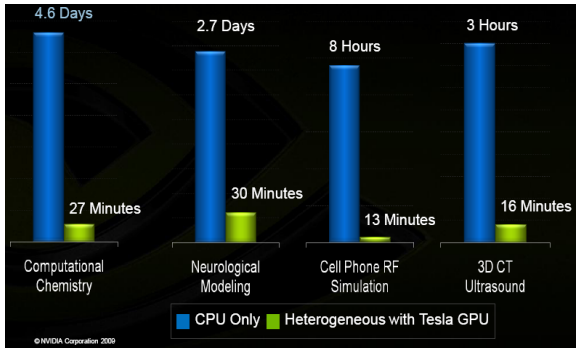
Operações aritméticas ++

Operações de memória --

GPUs

- Maior número de transístores para colocar em ULAs simplificadas
- Permite a realização de um maior número de cálculos ao mesmo tempo
- Controle de fluxo mais simples
- Aplicações paralelas onde as mesmas operações são aplicadas sobre um grande conjunto de dados
- Por que não usar esse poder de processamento em aplicações de propósito geral?????

Exemplos de aplicações



TESLA S2070

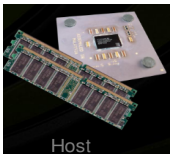
4 x 448 processadores	
Form Factor	1U
# of Tesla GPUs	4
GPU Memory Speed	1.55 GHz
GPU Memory Interface	384-bit
GPU Memory Bandwidth	148 GB/s
Double Precision floating point performance (peak)	2 Tflops
Single Precision floating point performance (peak)	4.13 Tflops
Total Dedicated Memory	12 GB GDDR5

Como Usar?

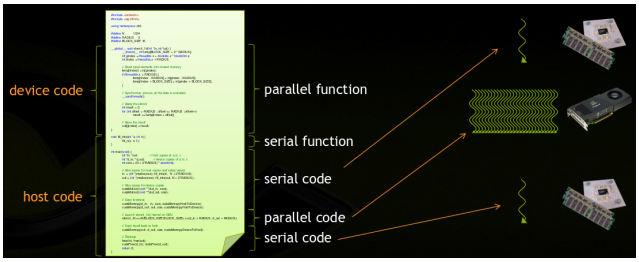
- Modelo *Heterogeneos Computing*
- CUDA
- OpenCL
- Direct Computer

Heterogeneos Computing Terminologia

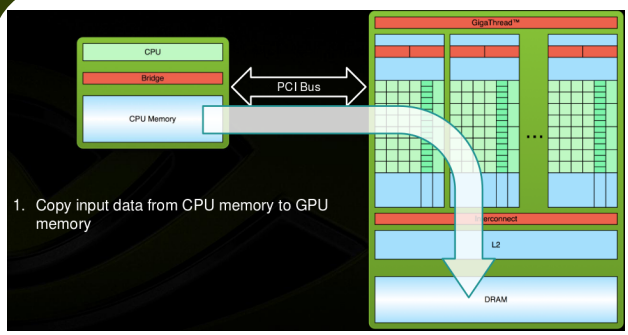
- Host
  - CPU
  - Memória
- Device
  - CPU
  - Memória (própria)



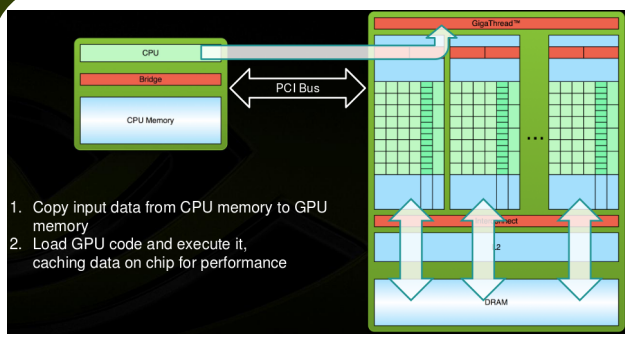
Heterogeneos Computing



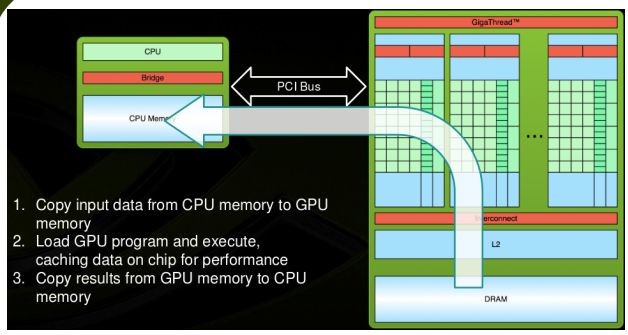
Funcionamento



Funcionamento

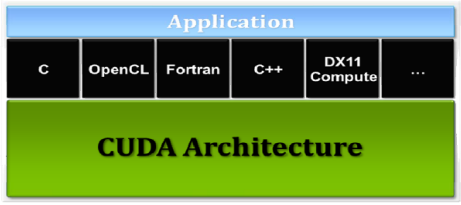


Funcionamento



CUDA

- *Compute Unified Device Architecture*
  - Arquitetura de computação paralela
  - Criada pela NVidia
  - Tirar proveito das GPUs (ou outros dispositivos)

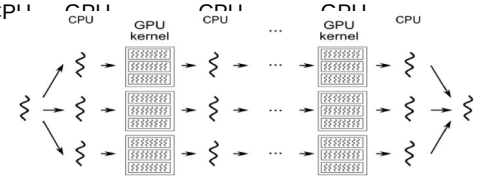


O que é CUDA

- Arquitetura paralela de propósito geral
- Objetivo: utilizar o poder computacional de GPUs nVIDIA
- Extensão da linguagem C que permite o uso de GPUs
  - Suporte a uma hierarquia de *threads*
  - Definição de *kernels* que são executados em GPUs
  - API com funções para gerenciamento da memória e outros tipos de controle

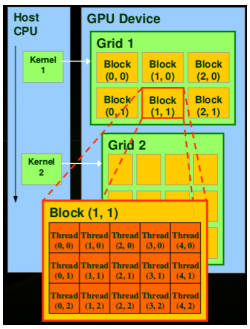
Modelo de Execução

- Execução do programa controlada pela CPU
  - Lança kernels executados na GPU
    - Trechos de código executados em paralelo por múltiplas threads
- Execução composta por 2 ciclos
  - CPU



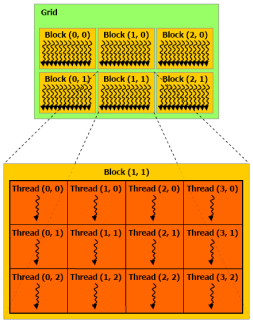
Grids e Thread Blocks

- As threads são organizadas em blocos, dentro de grids
- Um bloco (de threads) é um array de threads que cooperam por:
  - Memória Compartilhada
  - Sincronização
- Blocos de threads de um grid executam de forma independente



Exemplo Grid, Bloco, Threads

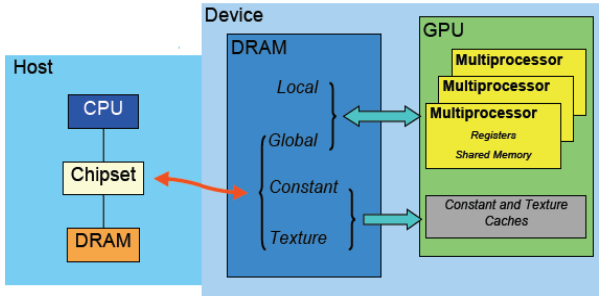
- Kernel executando em 72 threads
  - Grid 2D
    - Dimensão 3 x 2 x 1
    - 6 Blocos
  - Blocos 2D
    - Dimensão: 4 x 3 x1
    - 12 threads cada



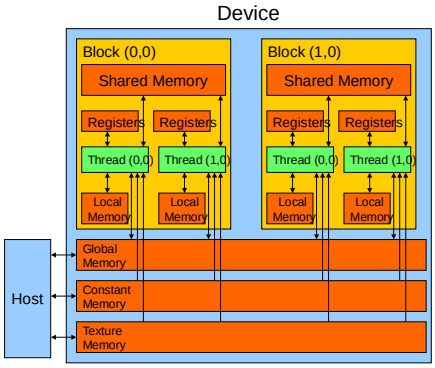
Modelo de memória

- Um device possui uma área de memória relativa ao device
- Cada Grid possui sua memória própria
  - Todas as threads do grid podem acessá-la
- Cada bloco possui um espaço de memória compartilhada
- Cada thread possui uma memória própria
  - Além dos registradores de controle

Modelo de Memória

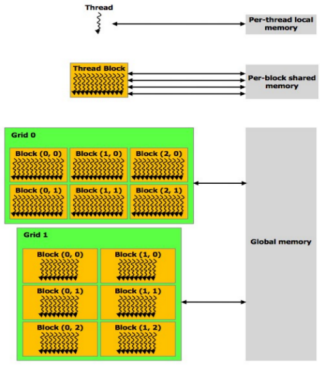


Modelo de memória



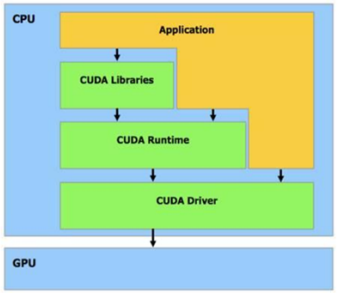
Memória e Threads

- Execução Kernel
  - Grid → Blocos → Threads
- Memória
  - Registradores por thread
  - Memória compartilhada por bloco
  - Memória Global



Arquitetura Software

- Programas em Cuda utilizam CUDA *Runtime*
  - Primitivas de alto nível
- É possível utilizar a API do *Driver*
  - Melhor controle da aplicação



Modelo de programação Threads

- Porque programar em Threads?
  - Para fazer distribuição de carga entre múltiplos núcleos
- Desafio de ter que manter o máximo de uso de cada núcleo
- CUDA permite até 12 mil* threads
  - CUDA é basicamente um cluster de threads

Modelo de programação Threads

- Em CPU
  - Poucas *threads* (troca contexto)
  - Natural gastar 1000 instruções para fazer a troca de uma thread para outra
- Em CUDA há outro paradigma....
- Não é necessário gerenciar as threads
  - Realizado em hardware
  - Sincronismo deve ser explicito

Modelo de Programação

- Kernel
  - Função geralmente escrita em C para CUDA
  - Executa no dispositivo N vezes em N **threads** em paralelo
- Blocos
  - São arranjos 1D, 2D ou 3D de threads
  - Cada thread de um Bloco possui um índice 1D, 2D ou 3D
  - Organizados em **grids**
- Grid é um arranjo 1D, 2D ou 3D de blocos
  - Cada bloco de um Grid possui um índice 1D 2D ou 3D

Identificando Threads e Blocos

- Threads e Blocos possuem um ID
  - Uso de *built-in variables*
- Blocos e threads
  - `blockIdx.x, blockIdx.y, blockIdx.z`
  - `threaIdx.x, threadIdx.y, threadIdx.z`
- Dimensões
  - `gridDim.x, gridDim.y, gridDim.z`
  - `blockDim.x, blockDim.y, blockDim.z`

Limites

- Cada *device* possui N *multiprocessing units*
- Cada multiprocessador pode executar um limite de threads
- Exemplo: 4 MP com 768 threads cada
- Limite = 4 * 768 = 3072 threads simultaneas
- Threads num bloco (x, y e z)
  - X * y * z <= 768
- Blocos ficam alocados no mesmo multiprocessador

Modelo de Programação

- Um kernel é uma função
  - Começa com o especificador **__global__**
  - Tem tipo de retorno **void**
- Kernels podem invocar funções
  - Especificadas como **__device__**
- Funções que executam no dispositivo
  - Não admitem número variável de argumentos
  - Não admitem recursão
  - Não admitem variáveis do tipo endereço de função

Invocando um kernel

```
kernel <<< #blocos, #threads >>> (argumentos)
```

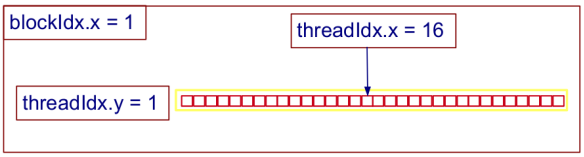
- Exemplos
  - nome_kernel <<< 1, 1 >>> (void)
    - Cria um bloco com apenas 1 thread = Σ threads = 1
  - kernel <<<2, 32>>>(void)
    - Cria 2 blocos com 32 threads cada
    - Σ threads = 64
- dim3 usado para especificar dimensões (x, y, z)
  - Dimensões não definidas = 1

Entendo Blocos e Threads

kernel<<< 1, 32 >>>( )

(1, 1, 1)

(32, 1, 1)



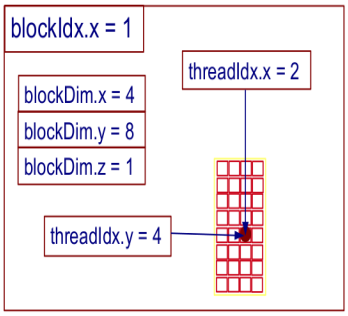
Entendo Blocos e Threads

dim3 bloco (4, 8);

kernel<<< 1, bloco >>>( )

(1, 1, 1)

(4, 8, 1)



Interface de Programação

- API Cuda Runtime e API de Driver
  - Gerência de memória do dispositivo
  - Transferência de dados entre host e device
  - Gerência de sistemas com vários dispositivos
  - . . . (Manual de referência CUDA)
- Runtime
  - Gerenciamento de Threads
  - Detecção de erros

Programa básico para CUDA

- Seleção do dispositivo a ser usado
  - **cudaSetDevice()**
- Alocação de memória no dispositivo
  - **cudaMalloc()**
- Transferência de dados entre host e dispositivo
  - **cudaMemcpy()**
- Liberação de memória no dispositivo
  - **cudaFree()**
- Finalização
  - **cudaThreadExit()**

Modelo de programa

```
__global__ void matrix_mul(…){
   …
   [GPU (Parallel) code]
   …
}

void main(){
   …
   [CPU (serial) code]
   …
   matrix_mul<<<dimGrid, dimBlock>>>(…)
   …
   [CPU (serial) code]
   …
}
```

Definição de um Kernel

Código na CPU

CUDA API Dispositivo

```
cudaError_t cudaGetDevice(int *device)
/* Returns in *device the current device for the
calling host thread. */

cudaError_t cudaSetDevice(int device)
/* Sets device as the current device for the
calling host thread. */

cudaError_t cudaGetDeviceCount(int *count)
/* Returns in *count the number of devices that
are available for execution. */
```

CUDA API Memória

```
cudaError_t cudaGetDeviceProperties(
        struct cudaDeviceProp *prop,
        int device)
/* Returns in *prop the properties of device dev. */
struct cudaDeviceProp {
   char name[256];
   size_t totalGlobalMem;
   size_t
sharedMemPerBlock;
   int regsPerBlock;
   int warpSize;
   int maxThreadsPerBlock;
   int maxThreadsDim[3];
   int maxGridSize[3];
   int multiProcessorCount;
   int computeMode;
   int concurrentKernels;
   int ECCEnabled;
   int pciBusID;
   int
maxThreadsPerMultiProcessor;
   }
```


CUDA API Memória

```
cudaError_t cudaMalloc(void ** devPtr,
                       size_t size )
/* Allocates size bytes of linear memory on
the device and returns in *devPtr a pointer
to the allocated memory. The memory is not
cleared. */

cudaError_t cudaFree(void *devPtr)
/* Frees the memory space pointed to by
devPtr, which must have been returned by a
previous call to cudaMalloc(). */
```


CUDA API Memória

```
cudaError_t cudaMemcpy(void *dst,
                       const void *src,
                       size_t count,
                       enum cudaMemcpyKind
kind)
/* Copies count bytes from the memory area
pointed to by src to the memory area
pointed to by dst.*/
kind: cudaMemcpyHostToHost
      cudaMemcpyHostToDevice
      cudaMemcpyDeviceToHost
      cudaMemcpyDeviceToDevice
```


Programa básico para CUDA

- Select Device to use
- Allocate host memory(malloc) for Array(s)
- Initialize host Array(s)
- Allocate device memory(cudaMalloc) for Array(s)
- Transfer data from host to device memory(cudaMemCpy)
- Specify kernel execution Configuaration
  - This is very important. Depending upon it blocks and threads automatically get assigned numbers
- Call Kernel
- Transfer result from device to host memory (cudaMemCpy)
- Deallocate host(free) and device(cudaFree) memories

Referências

Site NVIDEA, Cuda Zone

www.icmc.usp.br/~castelo/CUDA/slides2.ppt

https://eradsp2010.files.wordpress.com/2010/10/curso2_cuda_camargo.pdf

http://web.stanford.edu/class/ee380/Abstracts/080227-NickolIs-CUDAScalableParallelProgramming.pdf

Video com Uso de CUDA

http://developer.download.nvidia.com/presentations/2009/SIGGRAPH/Alternative_Rendering_Pipelines.mp4