

SuperComputação

Aula 16 – Introdução a GPU

2020 – Engenharia

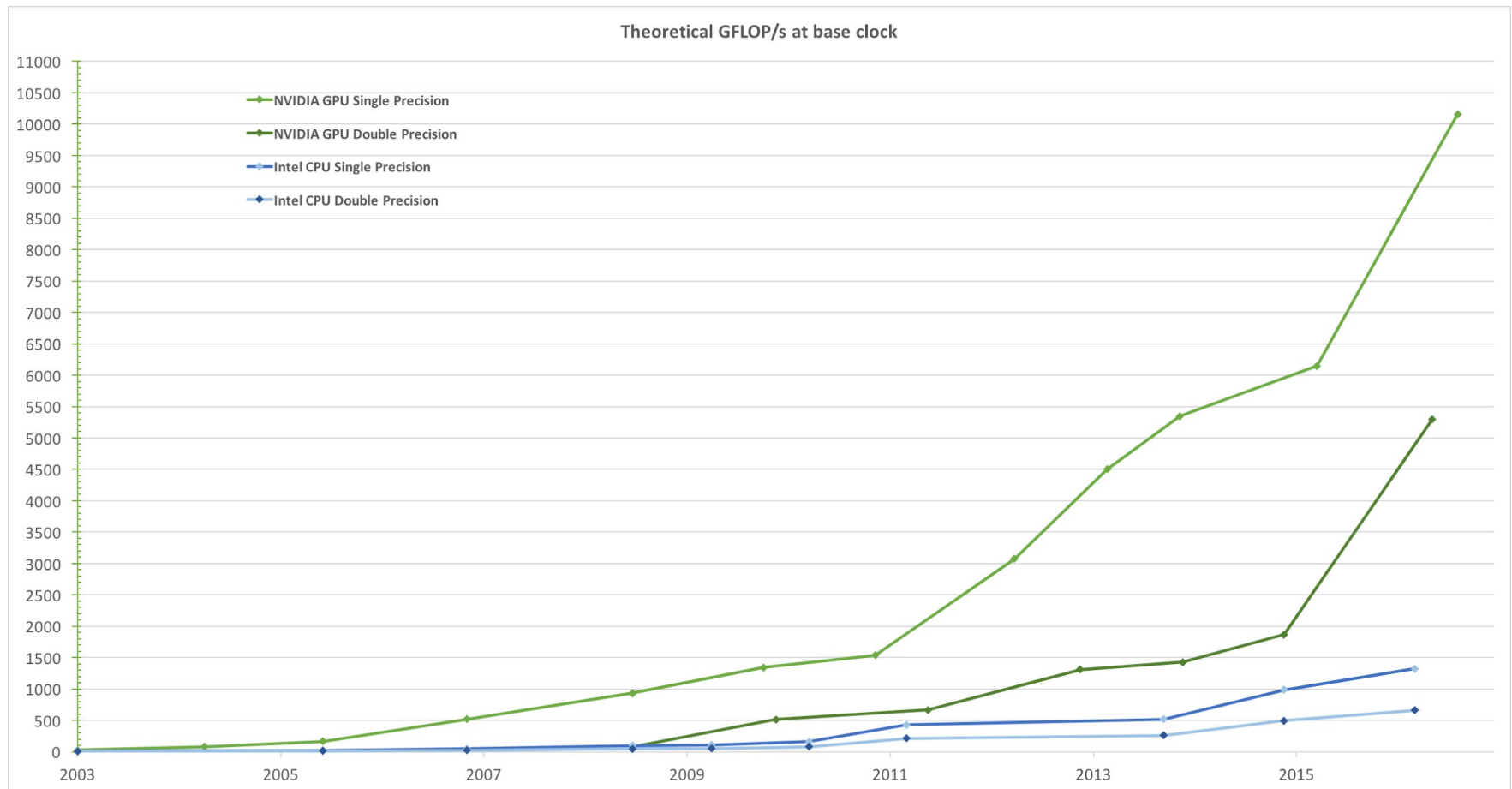
Luciano Soares <lpsoares@insper.edu.br>

Igor Montagner <igorsm1@insper.edu.br>

Objetivos de aprendizagem

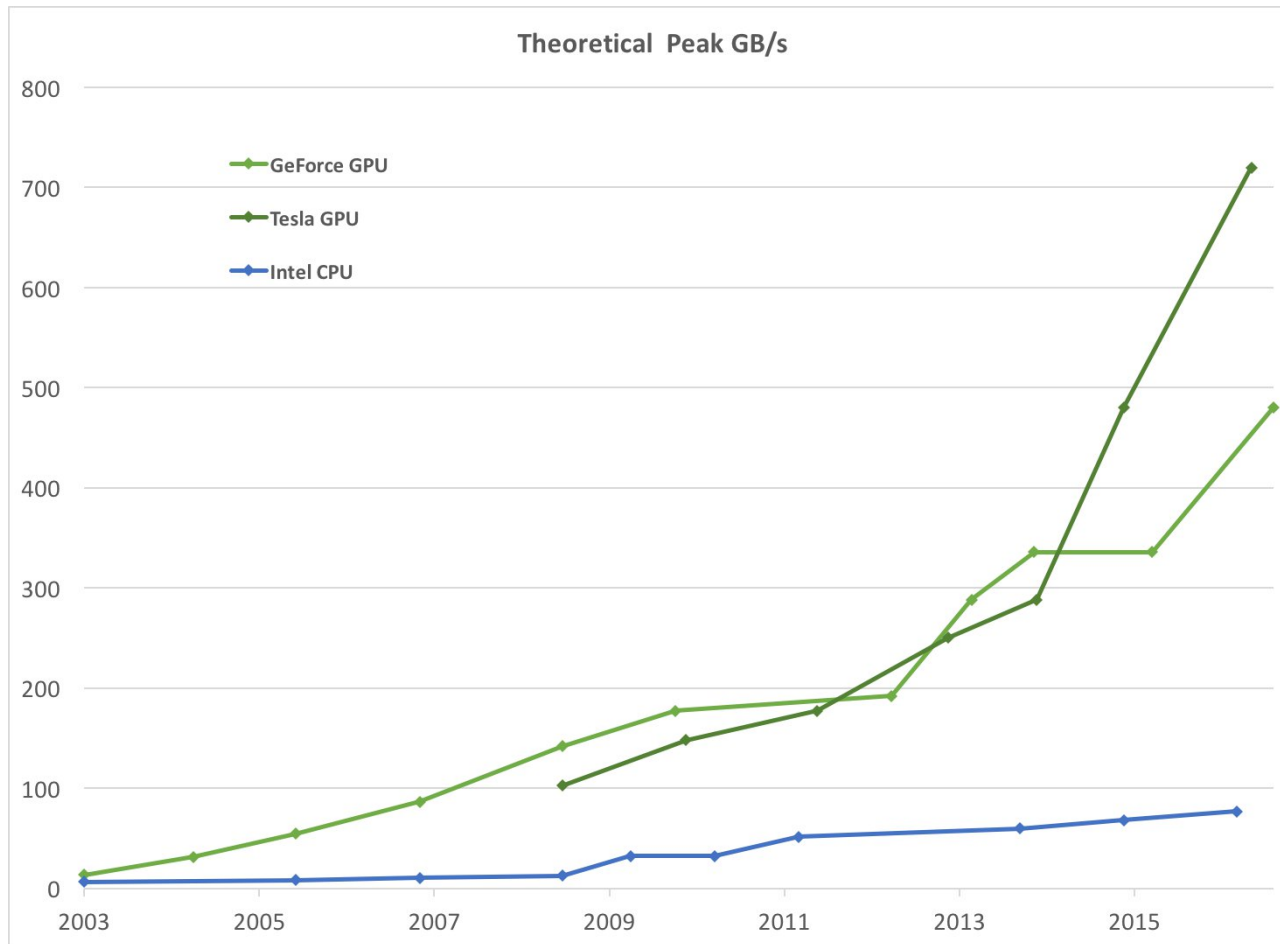
- Diferenciar dispositivos de latência (CPUs) e de throughput (GPU)
- Compreender layout de memória e transferência de dados em sistemas heterogêneos CPU-GPU
- Compilar primeiros programas na GPU

Desempenho em GFLOPS



Fonte: docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

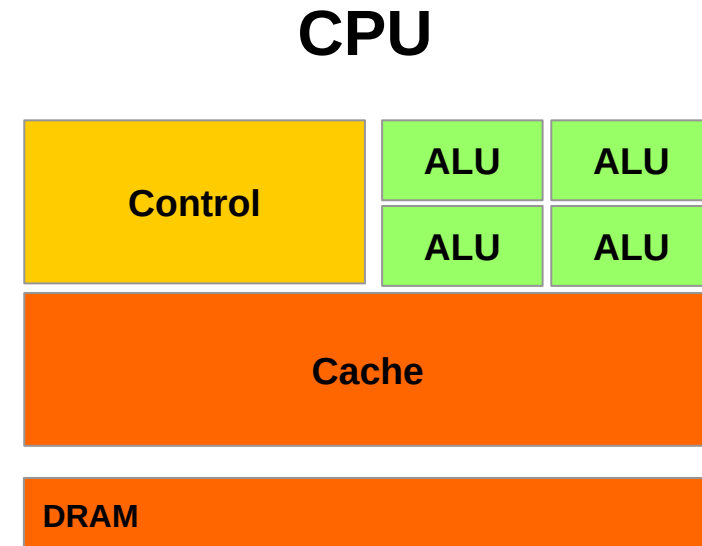
Desempenho em GB/s



Fonte: docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

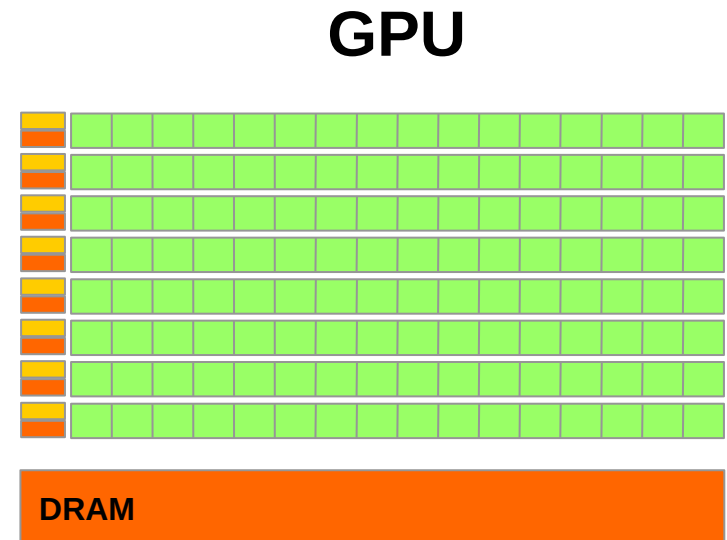
CPU minimiza latência

- ALU pontente minimiza latência das operações
- Cache grande:
 - Acelera operações lentas de acesso a RAM
- Controle sofisticado:
 - Branch prediction
 - Data forwarding



GPU minimiza throughput

- ALU simples
 - Eficiente energeticamente
 - Alta taxa de transferência
- Cache pequeno
 - Acesso contínuo a RAM
- Controle simples
- Número massivo de threads



CPU vs GPU

- CPUs para partes sequenciais onde uma latência mínima é importante
 - CPUs podem ser 10X mais rápidas que GPUs para código sequencial
- GPUs para partes paralelas onde a taxa de transferência(throughput) bate a latência menor.
 - GPUs podem ser 10X mais rápidas que as CPUs para código paralelo

Como usar GPU

Aplicações

Bibliotecas

Fácil de Usar
Alto Desempenho

Diretivas de
Compilação

Simple de Usar
Portabilidade de Código

Linguagens de
Programação

Maior Desempenho
Maior Flexibilidade

Bibliotecas aceleradas por GPU

Álgebra

Linear

FFT, BLAS,
SPARSE, Matrix

Numéricas/Mat
h

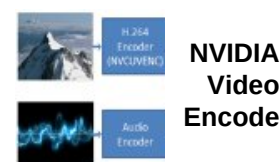
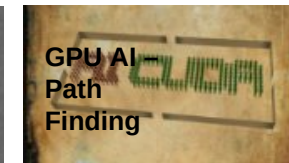
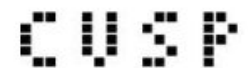
RAND, Estatísticas

Estrutura de
Dados/IA

Sort, Scan, Zero Sum

Processamento
Visual

Imagem & Video



Bibliotecas aceleradas por GPU

Facilidade de uso: O uso de bibliotecas permite a aceleração da GPU sem conhecimento aprofundado da programação da GPU

Simplicidade: Muitas bibliotecas aceleradas por GPU seguem APIs padrão, permitindo aceleração com mudanças mínimas de código

Qualidade: As bibliotecas oferecem implementações de alta qualidade de funções encontradas em uma ampla variedade de aplicativos

Linguagens de programação

Numerical analytics ▶

MATLAB, Mathematica, LabVIEW

Fortran ▶

CUDA Fortran

C ▶

CUDA C

C++ ▶

CUDA C++

Python ▶

PyCUDA, Copperhead, Numba

F# ▶

Alea.cuBase

Linguagens de programação

Desempenho: o programador tem o melhor controle de paralelismo e movimentação de dados

Flexível: a computação não precisa se encaixar em um conjunto limitado de padrões de biblioteca ou tipos de diretiva

Detalhado: O programador geralmente precisa expressar mais detalhes

Programando para GPU

- Compilador especial: nvcc
- Endereçamento de memória separado
 - Dados precisam ser copiados de/para GPU
 - Isto leva tempo
- Funções especiais (kernels) para rodar na GPU

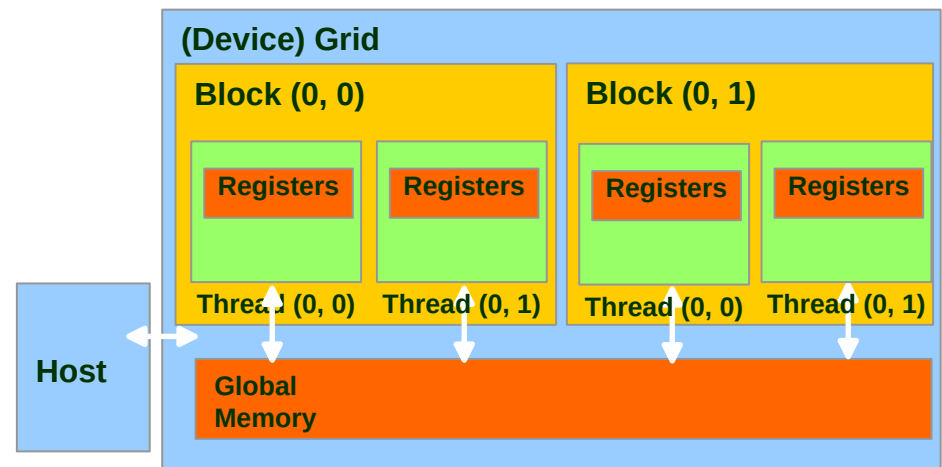
Memória em GPU

Código da GPU (device) pode:

- Cada thread ler e escrever nos **registradores**
- Ler e escrever na **memória global**

Código da CPU (host) pode:

- Transferir dados de e para **memória global**

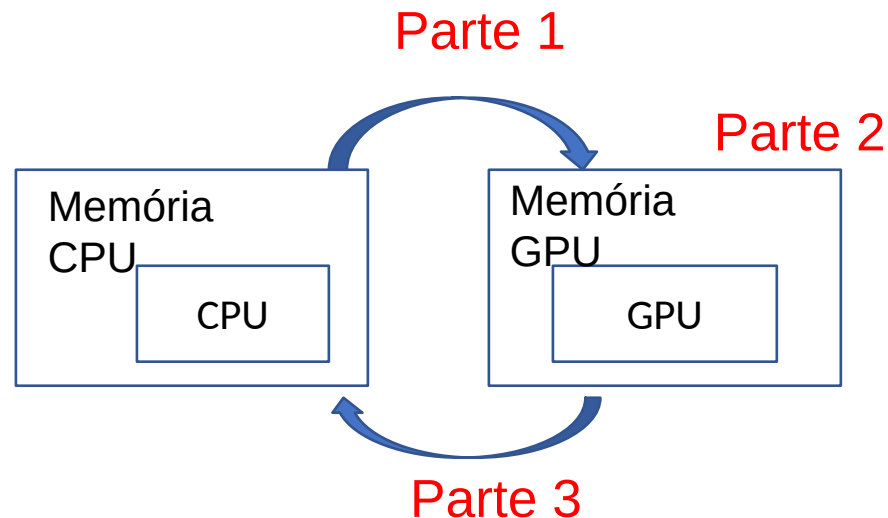


Fluxo de um programa

Parte 1: copia dados CPU → GPU

Parte 2: processa dados na GPU

Parte 3: copia resultados GPU → CPU



Programando para GPU - hoje



Nvidia Thrust: API simplificada em C++

Nvidia Thrust

Vantagens:

- **Simplifica transferências de memória**
- Duas operações customizáveis (*reduce, transform*)
- Suporta OpenMP e CUDA

Desvantagens:

- Limitado: menos recursos e desempenho que CUDA C
- Só tem dois tipos de operações
- Baseado em *templates* – *difícil de debugar erros de compilação*

Nvidia Thrust – tipos de dados

Apenas dois tipos

`thrust::device_vector<T>`

- Vetor genérico de dados na GPU
- Automaticamente alocado e desalocado
- Cópia é feita usando atribuição

`thrust::host_vector<T>`

- Vetor genérico de dados na CPU
- Pode ser substituído em vários lugares por contêineres da STL ou ponteiros “normais”

Nvidia Thrust – tipos de dados

```
thrust::host_vector<double> vec_cpu(10); // alocado na CPU  
  
vec1[0] = 20;  
vec2[1] = 30;  
  
thrust::host_vector<double> vec_gpu (10); // alocado na GPU
```

Alocação e cópias usando thrust

Nvidia Thrust – iteradores

Funcionam igual aos iteradores de `std::vector`

`v.begin()` // primeiro elemento

`v.end()` // último elemento

`v.begin()+2` // `v[2]`

`i = v.begin() + 3; *i = 4; // v[3] = 4`

Nvidia Thrust – iteradores

```
thrust::device_vector<int> v(5, 0); // vetor de 5 posições zerado
// v = {0, 0, 0, 0, 0}
thrust::sequence(v.begin(), v.end()); // inicializa com 0, 1, 2, ....
// v = {0, 1, 2, 3, 4}
thrust::fill(v.begin(), v.begin()+2, 13); // dois primeiros elementos = 3
// v = {13, 13, 2, 3, 4}
```

Nvidia Thrust – redução

Resume o vetor para um escalar

- Soma todos elementos
- Máximo/mínimo do vetor
- Contagens
- Suporta iteradores, o que torna a operação bastante flexível.

Nvidia Thrust – redução

```
val = thrust::reduce(iter_comeco, iter_fim, inicial, op);  
// iter_comeco: iterador para o começo dos dados  
// iter_fim: iterador para o fim dos dados  
// inicial: valor inicial  
// op: operação a ser feita.
```

Nvidia Thrust – transformação

Operações elemento a elemento entre pares de vetores ou um só vetor.

- Aritmética ponto a ponto
- Permite criação de operações customizadas
- Suporta iteradores de entrada e saída
- Funciona também para operações locais (imagens)

Nvidia Thrust – transformação

```
thrust::device_vector<double> V1(10, 0);
thrust::device_vector<double> V2(10, 0);
thrust::device_vector<double> V3(10, 0);
thrust::device_vector<double> V4(10, 0);
// inicializa V1 e V2 aqui

//soma V1 e V2
thrust::transform(V1.begin(), V1.end(), V2.begin(), V3.begin(), thrust::plus<double>());

// multiplica V1 por 0.5
thrust::transform(V1.begin(), V1.end(),
                  thrust::constant_iterator<double>(0.5),
                  V4.begin(), thrust::multiplies<double>());
```



Atividade prática

Iniciando com Thrust

1. Primeiros algoritmos em GPU

Insper

www.insper.edu.br