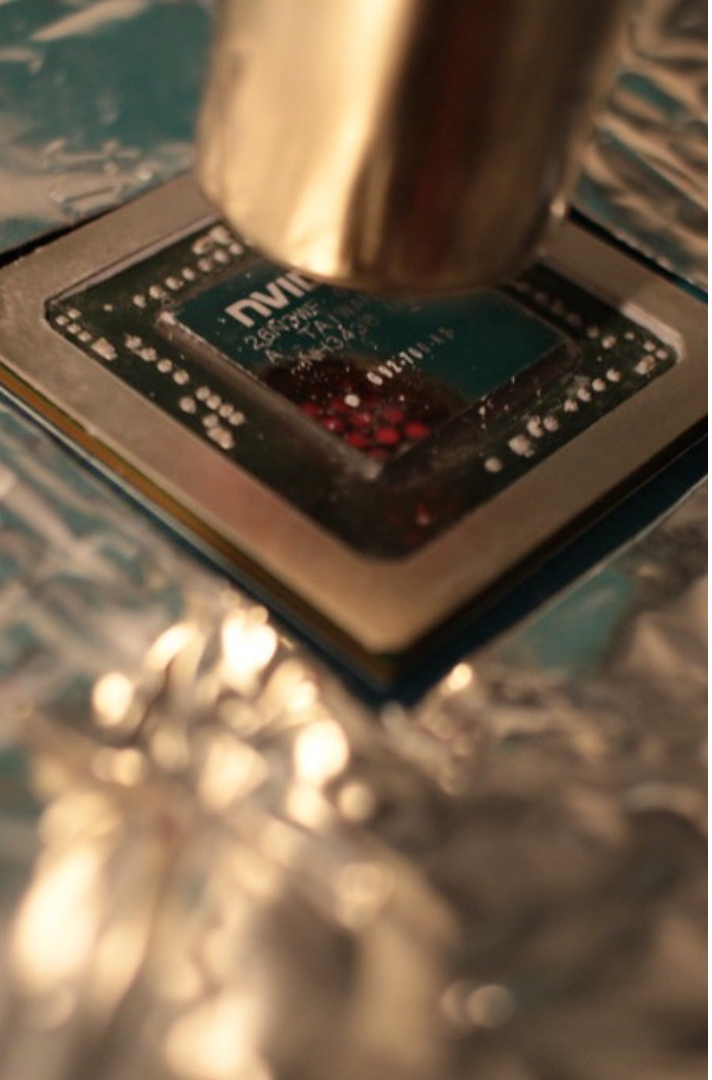
Insper

Supercomputação



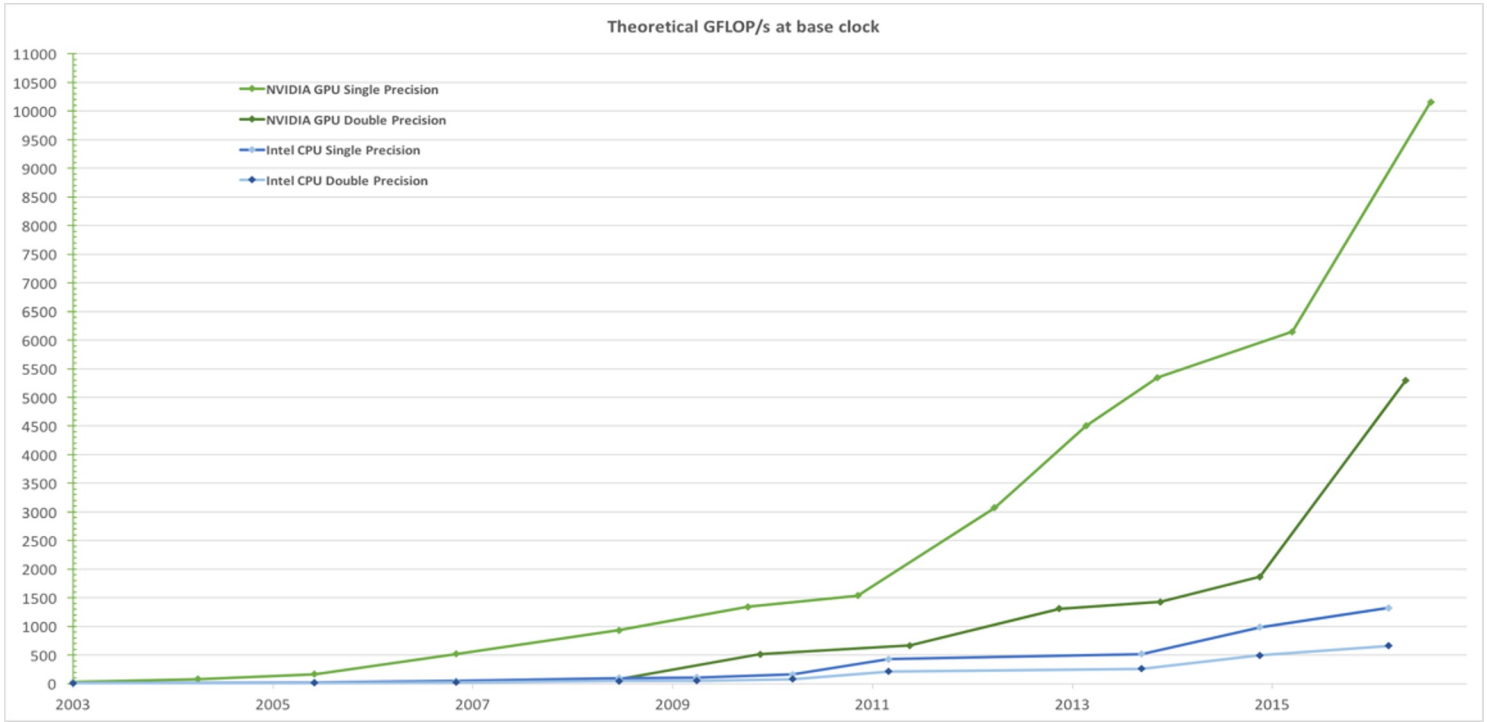
Aula - 17

- Introdução a GPU

Nossos objetivos

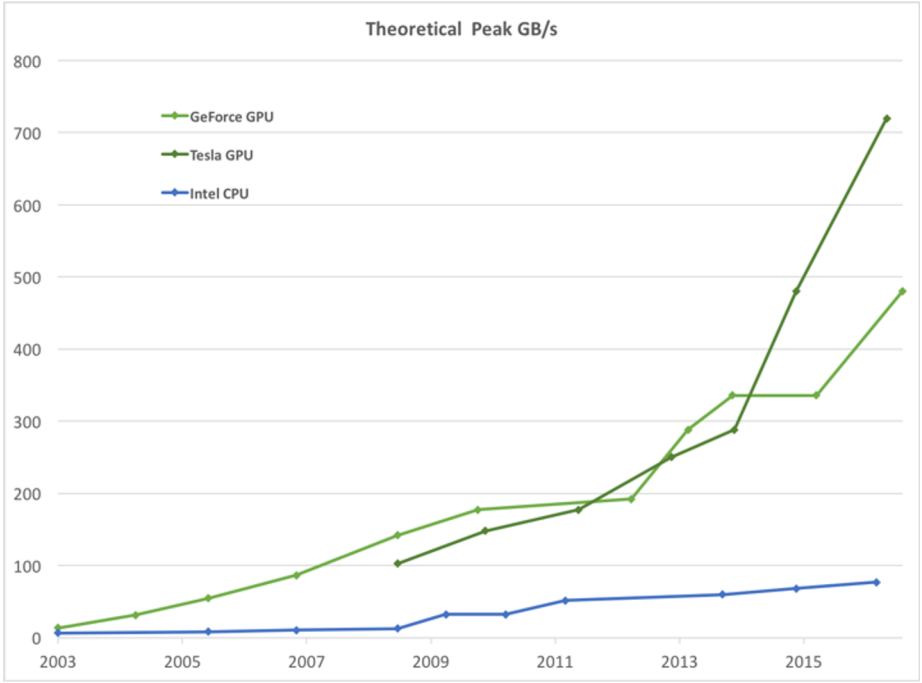
- Diferenciar dispositivos de latência (CPUs) e de throughput (GPUs)
- Compreender o layout de memória e transferência de dados em sistemas heterogêneos (CPU ⇔ GPU)
- Compilar primeiros programas na GPU

Desempenho em GFLOPS



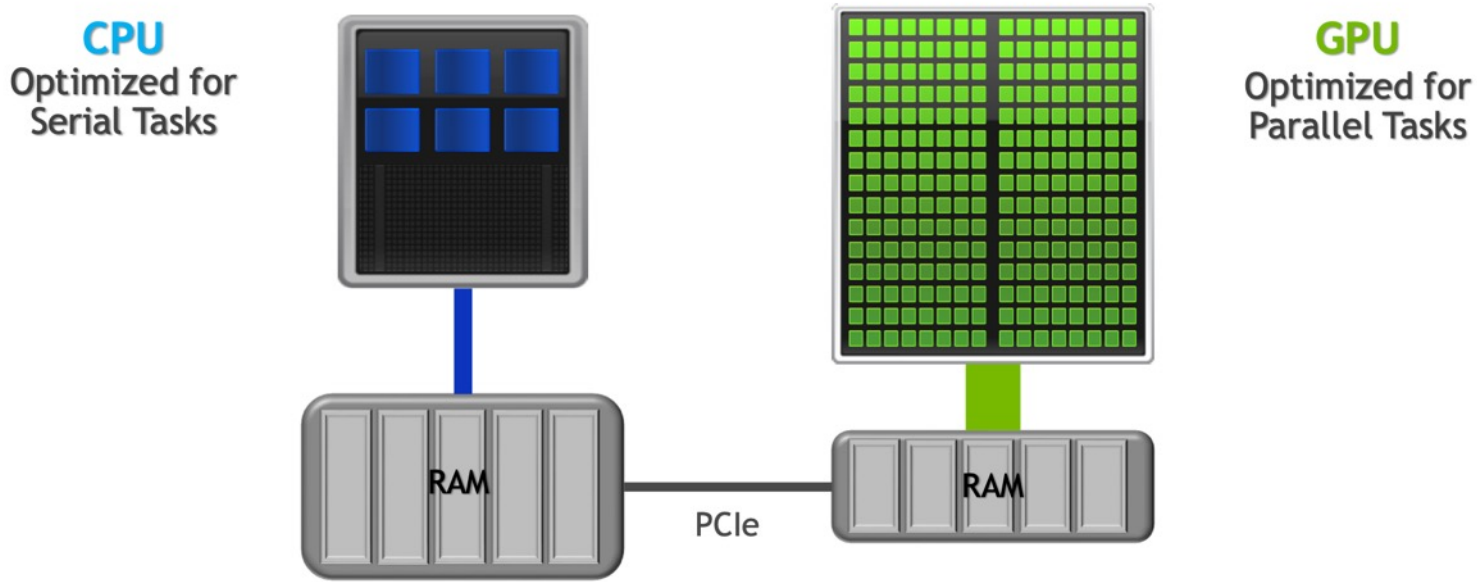
Fonte: docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Desempenho em GB/s



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

CPUs e GPUs


CPU
Optimized for
Serial Tasks
GPU
Optimized for
Parallel Tasks
RAM
PCIe
RAM

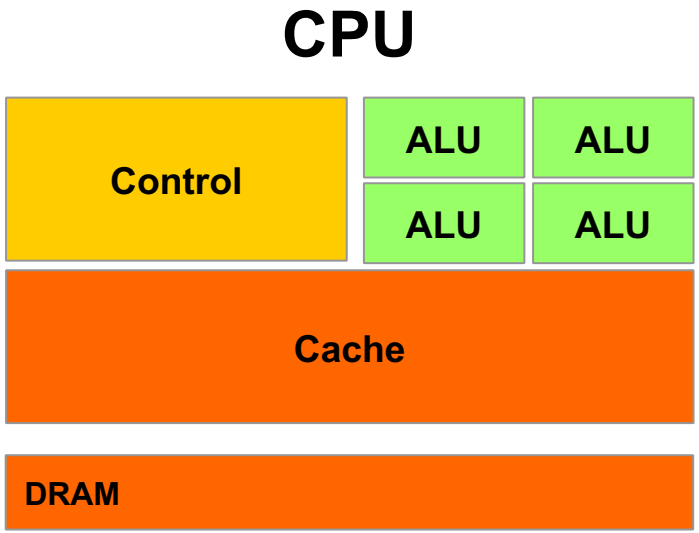
Speed vs Throughput

Velocidade

Throughput

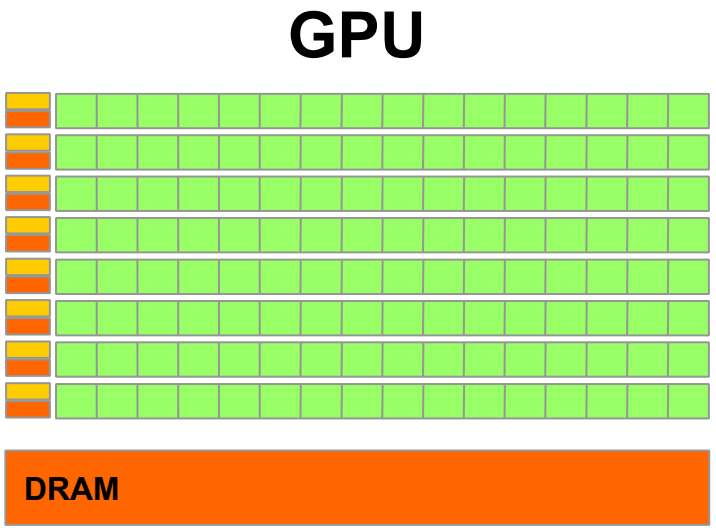
A CPU minimiza latência

- ULA potente, minimiza latência das operações
- Cache grande:
  - Acelera operações lentas de acesso à RAM
  - Mas cache-misses são custosos
  - Baixa performance / watt



A GPU maximiza throughput

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

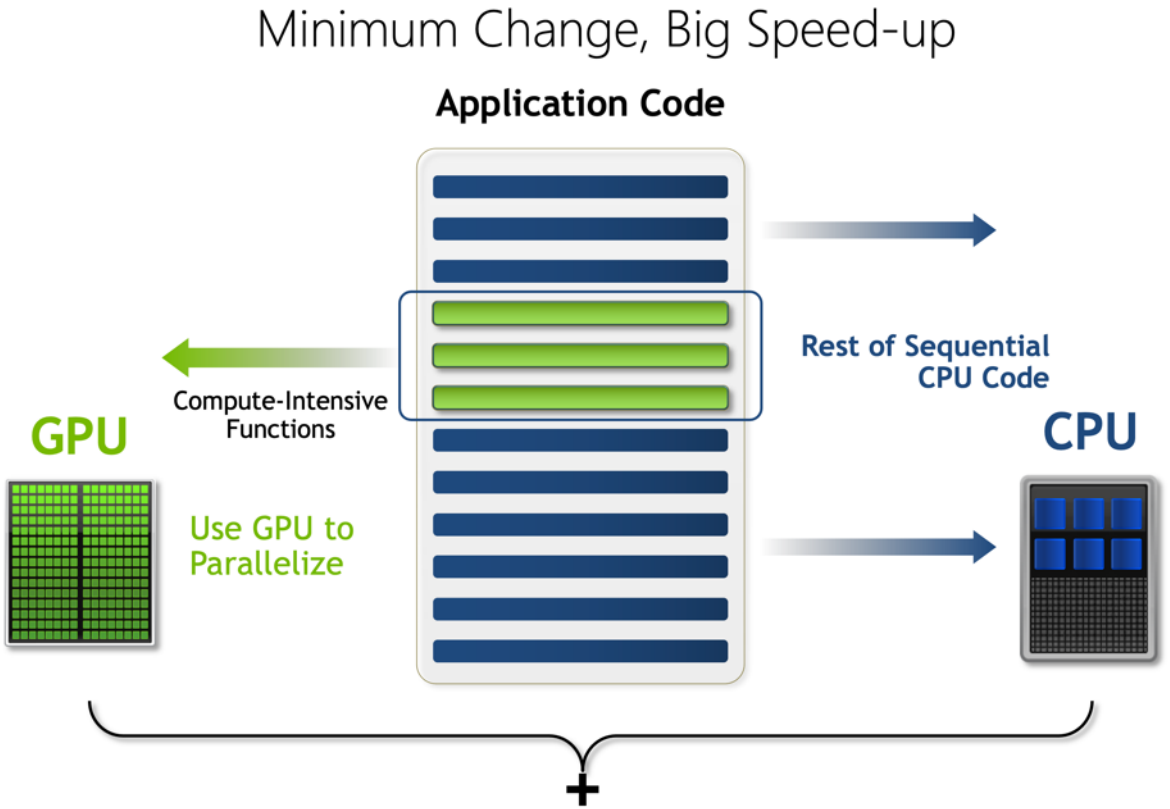


CPU vs GPU

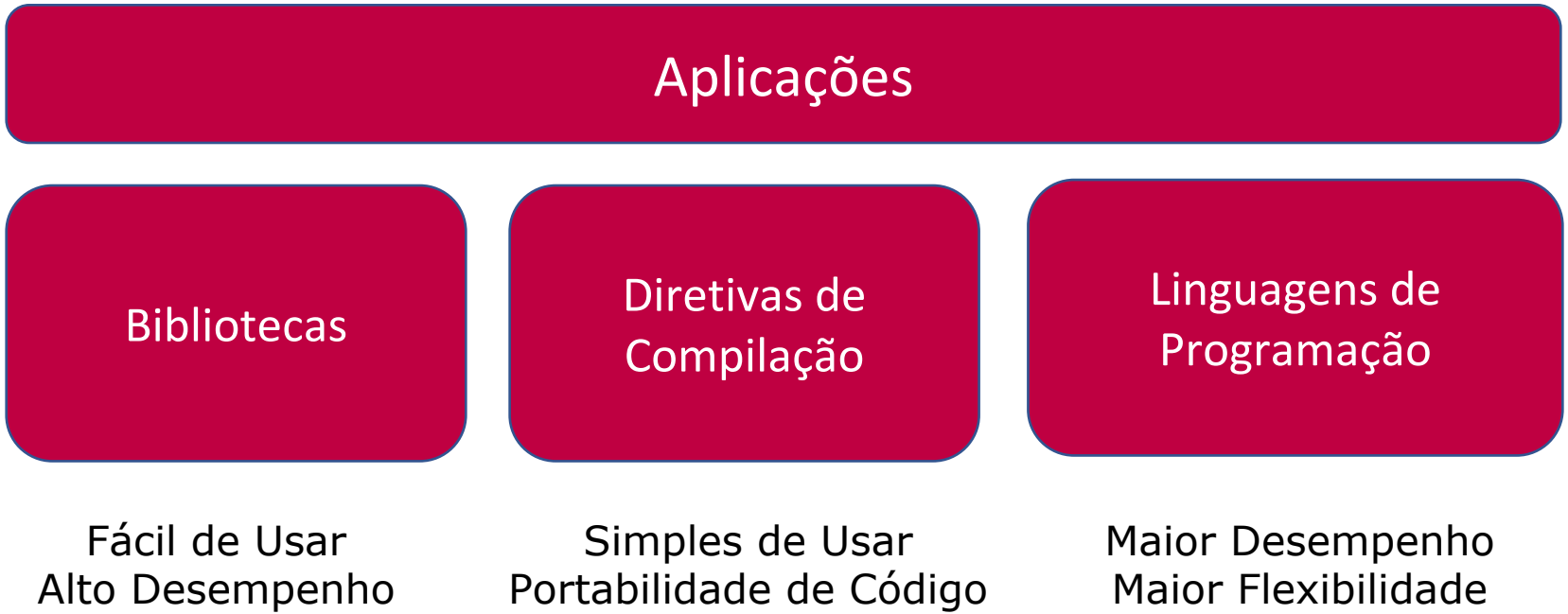
- CPUs para partes sequenciais onde uma latência mínima é importante
  - CPUs podem ser 10X mais rápidas que GPUs para código sequencial

- 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

CPU vs GPU


Minimum Change, Big Speed-up
Application Code
Compute-Intensive Functions
Rest of Sequential CPU Code
GPU
CPU
Use GPU to Parallelize

Como usar a GPU?



Bibliotecas aceleradas por GPU

Álgebra Linear
FFT, BLAS,
SPARSE, Matrix

NVIDIA cuFFT, cuBLAS, cuSPARSE

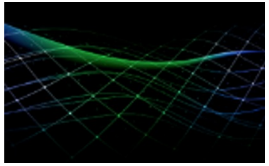
CULA tools

MAGMA
ICL UT

CUSP

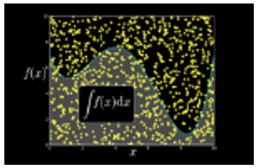
Numéricas/Math
RAND, Estatísticas

NVIDIA
Math Lib



ArrayFire

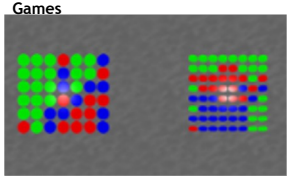
NVIDIA cuRAND



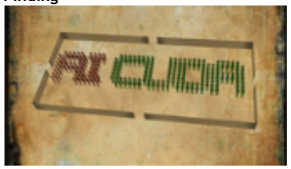
Estrutura de Dados/IA
Sort, Scan, Zero Sum

Thrust

GPU AI - Board
Games

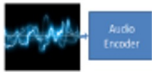


GPU AI - Path
Finding



Processamento Visual
Imagem & Video

NVIDIA
NPP



NVIDIA
Video Encode

Sundog
Software

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

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

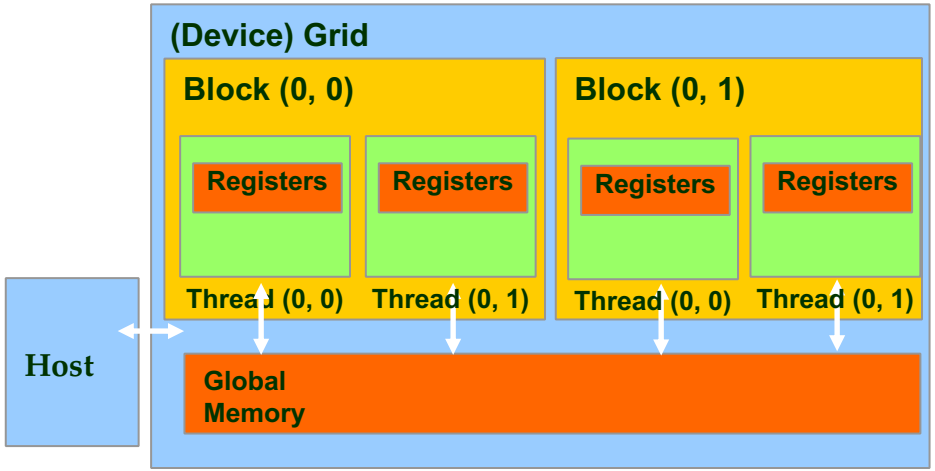
Memória em GPUs

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

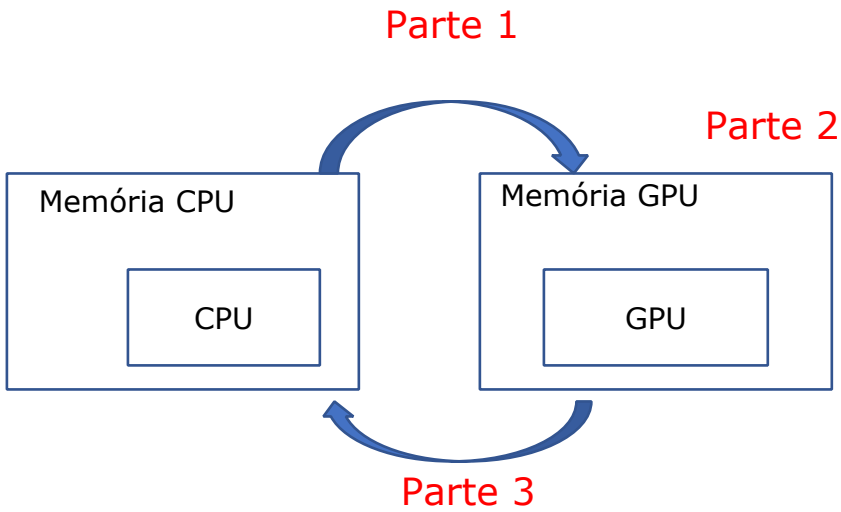


Fluxo dos programas

Parte 1: copia dados CPU → GPU

Parte 2: processa dados na GPU

Parte 3: copia resultados GPU → CPU



Biblioteca que usaremos - Thrust

Importante: infra

Se você tem uma GPU:

- pode usá-la diretamente na disciplina;
- instale o pacote nvidia-cuda-toolkit e os drivers compatíveis

Se você não tem GPU:

- compile código usando thrust/OpenMP
- solicite a conversão da sua VM para GPU com o Tiago (tiagoaodc@insper.edu.br)

A partir da próxima semana vamos supor que todos já tem acesso a uma GPU com compilador funcionando.

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 containers da STL ou ponteiros “normais”

Thrust - Exemplo

```
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

vec_gpu = vec_cpu; // copia o conteúdo da CPU para GPU
thrust::device_vector<double> vec2_gpu (vec_cpu); // também transfere para GPU
```

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
```

Thrust - Exemplo

```
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}
```

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.

Thrust - Exemplo

```
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.
```

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)

Thrust - Exemplo

```
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>());
```

Thrust - Roteiro

Iniciando com Thrust

- Alocação de memória em CPU e GPU
- Utilização das operações de redução e transformação
- Operações elemento e elemento disponíveis na thrust

Thrust

- Integração numérica
- Iterators

Simple Numerical Integration: Example

```
thrust::device_vector<int> width(11, 0.1);
width       =  0.1   0.1   0.1   0.1   0.1   0.1   0.1   0.1   0.1   0.1   0.1

thrust::sequence(x.begin(), x.end(), 0.0f, 0.1f);
x           =  0.0   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1.0

thrust::transform(x.begin(), x.end(), height.begin(), square());
height      =  0.0  0.01  0.04  0.09  0.16  0.25  0.36  0.49  0.64  0.81   1.0

thrust::transform(width.begin(), width.end(), height.begin(), area.begin(),
thrust::multiplies<float>())
area        =  0.0 0.001 0.004 0.009 0.016 0.025 0.036 0.049 0.064 0.081   0.1

total_area = thrust::reduce(area.begin(), area.end());
total_area =  0.385


thrust::inclusive_scan(area.begin(), area.end(), accum_areas.begin());
accum_areas =  0.0 0.001 0.005 0.014 0.030 0.055 0.091 0.140 0.204 0.285 0.385
```

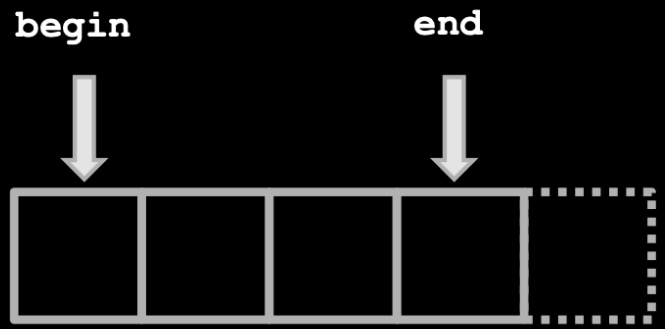
Iterators

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

thrust::device_vector<int>::iterator begin = d_vec.begin();
thrust::device_vector<int>::iterator end   = d_vec.end();

int length = end - begin; // compute size of sequence [begin, end)

end = d_vec.begin() + 3;  // define a sequence of 3 elements
```



Constant Iterators

- `constant_iterator`
  - Mimics an infinite array filled with a constant value

```
// create iterators
constant_iterator<int> begin(10);
constant_iterator<int> end = begin + 3;

begin[0]   // returns 10
begin[1]   // returns 10
begin[100] // returns 10

// sum of [begin, end)
reduce(begin, end);   // returns 30 (i.e. 3 * 10)
```

Counting Iterator

- counting_iterator
  - Mimics an infinite array with sequential values

```
// create iterators
counting_iterator<int> begin(10);
counting_iterator<int> end = begin + 3;

begin[0]   // returns 10
begin[1]   // returns 11
begin[100] // returns 110

// sum of [begin, end)
reduce(begin, end);   // returns 33 (i.e. 10 + 11 + 12)
```

Zip Iterator

zip_iterator

```
// initialize vectors
device_vector<int>  A(3);
device_vector<char> B(3);
A[0] = 10;  A[1] = 20;  A[2] = 30;
B[0] = 'x'; B[1] = 'y'; B[2] = 'z';

// create iterator (type omitted)
begin = make_zip_iterator(make_tuple(A.begin(), B.begin()));
end   = make_zip_iterator(make_tuple(A.end(),   B.end()));

begin[0]   // returns tuple(10, 'x')
begin[1]   // returns tuple(20, 'y')
begin[2]   // returns tuple(30, 'z')

// maximum of [begin, end)
maximum< tuple<int,char> > binary_op;
reduce(begin, end, begin[0], binary_op); // returns tuple(30, 'z')
```

Five Operations You Can Do with a Lot of Data in Parallel

- Generate/Create
  - Automatically fill with programmatically defined data
- Transform
  - Apply some “operation” to each element of the data
  - Also called “Map” in many contexts
- Compact
  - Take only the elements in which you are interested
  - Also called “Filter” in many contexts
- Expand
  - The opposite of Compact
  - Create a larger data set from a smaller data set
- Aggregate
  - Calculate a “summary” of your data (e.g., sum or average)
  - Also called “Reduce” or “Fold”
  - “Scan” also provides all intermediate values

Simple Examples with Thrust Pseudocode

- **Generate**

```
thrust::sequence(0,4)   0  1  2  3  4
```

- **Transform**

```
input                   4  5  2  1  3
thrust::transform(+1)   5  6  3  2  4
```

- **Compact**

```
input                   4  5  2  1  3
thrust::copy_if(even)   4  2
```

- **Expand**

```
input                   4  5  2  1  3
thrust::for_each(x,2x)  4  8  5 10  2  4  1  2  3  6
```

- **Aggregate**

```
input                   4  5  2  1  3
thrust::reduce(+)       15
```

Generate Data in Parallel

- Many copies of a certain constant value
  - ```
    // Ten elements with initial value of integer 1
    thrust::device_vector<int> x(10, 1);
    ```
- A sequence of increasing or decreasing values
  - ```
    // Allocate space for ten integers, uninitialized
    thrust::device_vector<int> y(10);
    // Fill the space with integers
    thrust::sequence(y.begin(), y.end(), 5, 2);
    ```
- Random Values
  - Multiple copies of a random number generator
  - Give each one a different seed

Transform: Vector Addition

Apply a binary operator “plus” to each element in x and y

```
- thrust::transform(x.begin(), x.end(), // begin and end of the
                                        // first input vector
  y.begin(),                            // begin of the second
                                        // input vector
  result.begin(),                       // begin of the result
                                        // vector
  thrust::plus<int>());                 // predefined integer
                                        // addition
```

```
-       x: 1  1  1  1  1  1  1  1  1  1
                   +
        y: 5  7  9 11 13 15 17 19 21 23
                   =
  result: 6  8 10 12 14 16 18 20 22 24
```

Transform: Uniform Sampling of a Mathematical Function

- Q: How are we going to generate something more complicated?
  A: Start from some sequence and apply some transformation
- Sampling the function $f(x) = x^2$ in the interval of [0, 1]

```
// Generate a sequence of x_i in [0,1] with dx=0.1
// in between each of them
float dx = 1.0f/10.0f;
thrust::sequence(x.begin(), x.end(), 0.0f, dx);

// Apply the square operation to each of the x_i
// to transform into f(x_i) = y_i = x_i^2
thrust::transform(x.begin(), x.end(),
                  y.begin(),
                  square());
```

```
x: 0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.0
y: 0.0  0.01 0.04 0.09 0.16 0.25 0.36 0.49 0.64 0.81 1.0
```

Simple Numerical Integration: Example

```
thrust::device_vector<int> width(11, 0.1);
width       =  0.1   0.1   0.1   0.1   0.1   0.1   0.1   0.1   0.1   0.1   0.1

thrust::sequence(x.begin(), x.end(), 0.0f, 0.1f);
x           =  0.0   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1.0

thrust::transform(x.begin(), x.end(), height.begin(), square());
height      =  0.0  0.01  0.04  0.09  0.16  0.25  0.36  0.49  0.64  0.81   1.0

thrust::transform(width.begin(), width.end(), height.begin(), area.begin(),
thrust::multiplies<float>())
area        =  0.0 0.001 0.004 0.009 0.016 0.025 0.036 0.049 0.064 0.081   0.1

total_area = thrust::reduce(area.begin(), area.end());
total_area =  0.385


thrust::inclusive_scan(area.begin(), area.end(), accum_areas.begin());
accum_areas =  0.0 0.001 0.005 0.014 0.030 0.055 0.091 0.140 0.204 0.285 0.385
```

Reduce: Simple Numerical Integration

Apply what we learned to estimate the area under a curve

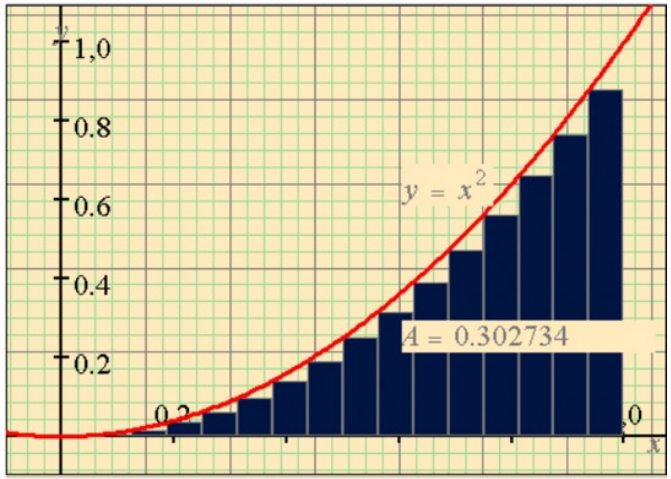
Create a constant vector of widths

Create a vector of heights from the function values

Apply multiply operation on each element of width and height

Sum all the computed areas to get the total area

In calculus, this is a method of estimating the integral



$$\int_0^1 f(x)dx \approx \sum_{i=1}^{n} f(x_i)\Delta x$$

Scan: Simple Numerical Integration

- What happens if we are interested in the integral $F(t) = F(0) + \int_0^t f(x)dt$

  on the interval [0, 1] instead of just a number?

- Calculate a running sum by using scan

- ```
  thrust::inclusive_scan(y_dx.begin(), y_dx.end(),
                         F.begin());
  ```

- ```
  f(x_i)*dx =  0.0 0.001 0.004 0.009 0.016 0.025 0.036 0.049 0.064 0.081   0.1
  F(t)      =  0.0 0.001 0.005 0.014 0.030 0.055 0.091 0.140 0.204 0.285 0.385
  ```

- The last element of the scan (0.385) is the same as the output of reduce

- In mathematical terms,

$$\int_0^1 f(x)dx = F(1) - F(0)$$

Insp

Usando CUDA no Colab

- https://www.geeksforgeeks.org/how-to-run-cuda-c-c-on-jupyter-notebook-in-google-colaboratory/