

Programação para GPU usando CUDA

Alvaro Alvin Oesterreich Santos

Ricardo de la Rocha Ladeira

Introdução

Em busca de aumentar o desempenho de programas, é comum buscar paralelizar partes da execução para permitir o uso simultâneo de múltiplos processadores de uma CPU (*Central Processing Unit*). CPUs a nível de consumidor atuais possuem de um a 92 núcleos [1], com CPUs para servidores chegando até 128 núcleos [2].

Entretanto, existem aplicações que podem obter vantagem com o uso simultâneo de ainda mais processadores e a GPU (*Graphics Processing Unit*) ou GPGPU (*General Purpose Graphics Processing Unit*) quando não é utilizada para aplicações gráficas, são placas especiais que podem conter até 16 mil processadores, aproximadamente [3].

Existem alguns modelos de computação que permitem que sejam executados códigos de propósito geral em GPUs, transformando-as em GPGPUs. Um dos modelos de computação que permite isso é desenvolvido pela NVIDIA e se chama CUDA¹ (*Compute Unified Device Architecture*). CUDA permite escrever programas que são executados na CPU e na GPGPU de forma conjunta.

Processamento em GPGPU

Inicialmente, GPUs foram feitas para fazer processamento gráfico, por isso são adequadas para processamentos do tipo SIMD (*Single Instruction Multiple Data*). Elas costumam possuir muito mais processadores do que CPUs, mas executam em um *clock* mais baixo.

As GPUs são otimizadas para fazerem o processamento de conjuntos de dados que requerem computação semelhante, de forma que possam ser processados de forma paralela. Por isso processamentos que podem se adequar a essas características podem

¹ <https://developer.nvidia.com/cuda-toolkit>

ter grande ganho de desempenho, por permitir ser executadas em uma grande quantidade de processadores, a computação não relacionada a gráficos em uma GPU a torna uma GPGPU.

Entretanto, a GPGPU possui algumas limitações devido a sua arquitetura. Ela possui uma grande quantidade de ALUs (*Arithmetic and Logic Units*) que, por serem especializadas em operações de ponto flutuante, não dão suporte nativo a operações *bit shift* e *bitwise*. Inicialmente não existia suporte nativo a operações de inteiros e aritmética de ponto flutuante de precisão dupla, mas as arquiteturas mais recentes fornecem essas operações sem necessidade de emulação. A Tabela 1 mostra uma comparação das características da CPU e da GPU.

Tabela 1. Comparação entre CPU e GPU.

CPU	GPU
Caches muito rápidas (bom para reúso de dados)	Muitas unidades Lógicas e Aritméticas
Granularidade de ramificação	Acesso rápido à memória interna
Muitos <i>threads</i> /processos diferentes	Executa um programa em cada fragmento/ <i>vertex</i>
Alta performance em execuções em um único <i>thread</i>	Alta taxa de transferência de dados em tarefas paralelas
Boa para paralelização de tarefas	Boa para paralelização de dados
Otimizada para alto desempenho em códigos sequenciais (<i>caches</i> e previsão de ramificação)	Otimizada para alta intensidade de operações aritméticas de natureza paralela (operações de ponto flutuante)

Fonte: Adaptado de [4].

CUDA

Existem vários SDKs (*Software Development Kit*) e várias APIs (*Application Programming Interface*) que permitem fazer o processamento em GPGPUs, tais como CUDA, ATI Stream SDK, OpenCL, Rapidmind, HMPP e PGI Accelerator. Um dos mais utilizados é o CUDA, desenvolvido e mantido pela NVIDIA. Até o momento, o CUDA funciona somente com

placas NVIDIA por possuírem os chamados CUDA *cores*. A Figura 1 apresenta a arquitetura de uma GPU compatível com CUDA.

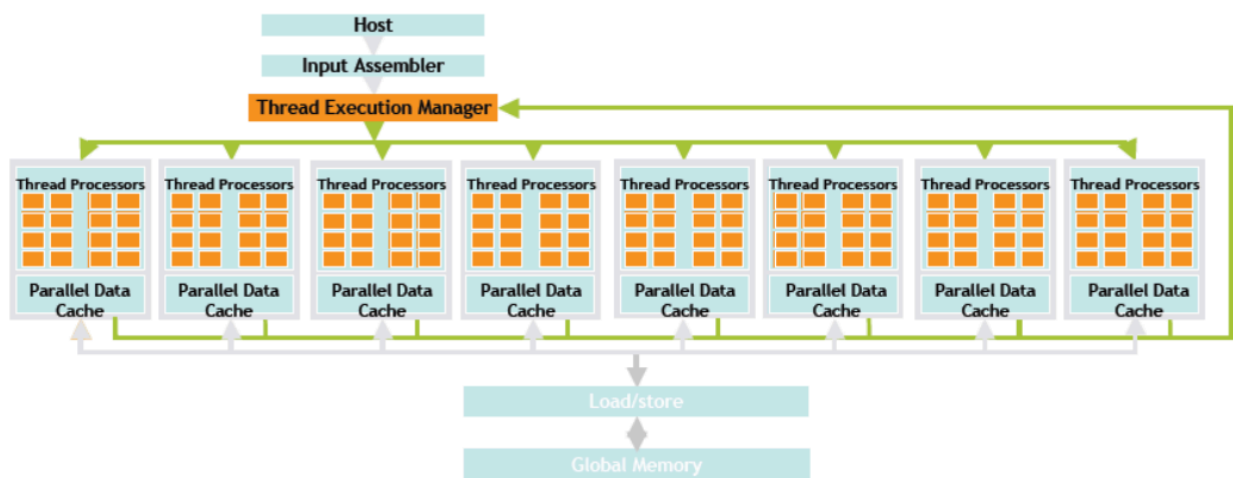


Figura 1. Arquitetura de uma GPU compatível com CUDA. Fonte: [5].

Como é possível ver na Figura 1, a GPU é composta por um fluxo de processadores que possuem grande capacidade de paralelismo. Cada GPU possui diversos *grids*, indicados por retângulos de cor azul clara, conectados por uma flecha verde na Figura 1. Cada *grid* possui blocos, indicados pelos retângulos laranjas contidos dentro de cada *grid*, e cada um desses blocos possui múltiplas *threads*, que não estão exibidas na figura. A Figura 2 exhibe os componentes de processamento da GPU.

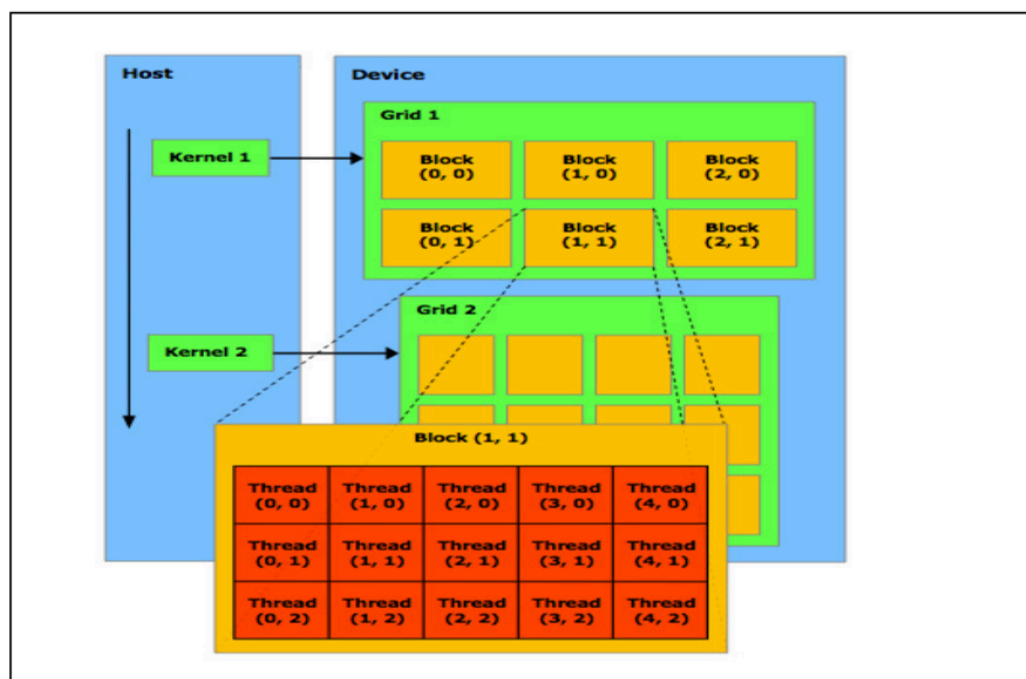


Figura 2. Visão mais detalhada dos componentes de processamento da GPU. Fonte: [5].

As funções executadas pela GPU são chamadas de *kernels*. CUDA passa cada *kernel* para um *grid* e todas as *threads* que estão naquele *grid* executam o mesmo *kernel* de forma assíncrona. Entretanto, cada *grid*, cada bloco e cada *thread* possui seu próprio identificador referente à estrutura que está contida, dessa forma é possível determinar qual dado cada *thread* deverá processar.

Com relação a memória, cada *thread* possui uma memória de acesso rápido privada e também pode acessar uma memória que é compartilhada pelas *threads* de um bloco. A partir desse nível é preciso se preocupar com condições de corrida. Cada *grid* também tem acesso a uma memória global que é compartilhada por todos os *grids*. As arquiteturas mais recentes dão suporte à memória unificada que permite que a memória da GPU e da CPU possam ser acessadas através de um único endereçamento. A Figura 3 apresenta um diagrama que indica o acesso à memória de cada parte do sistema.

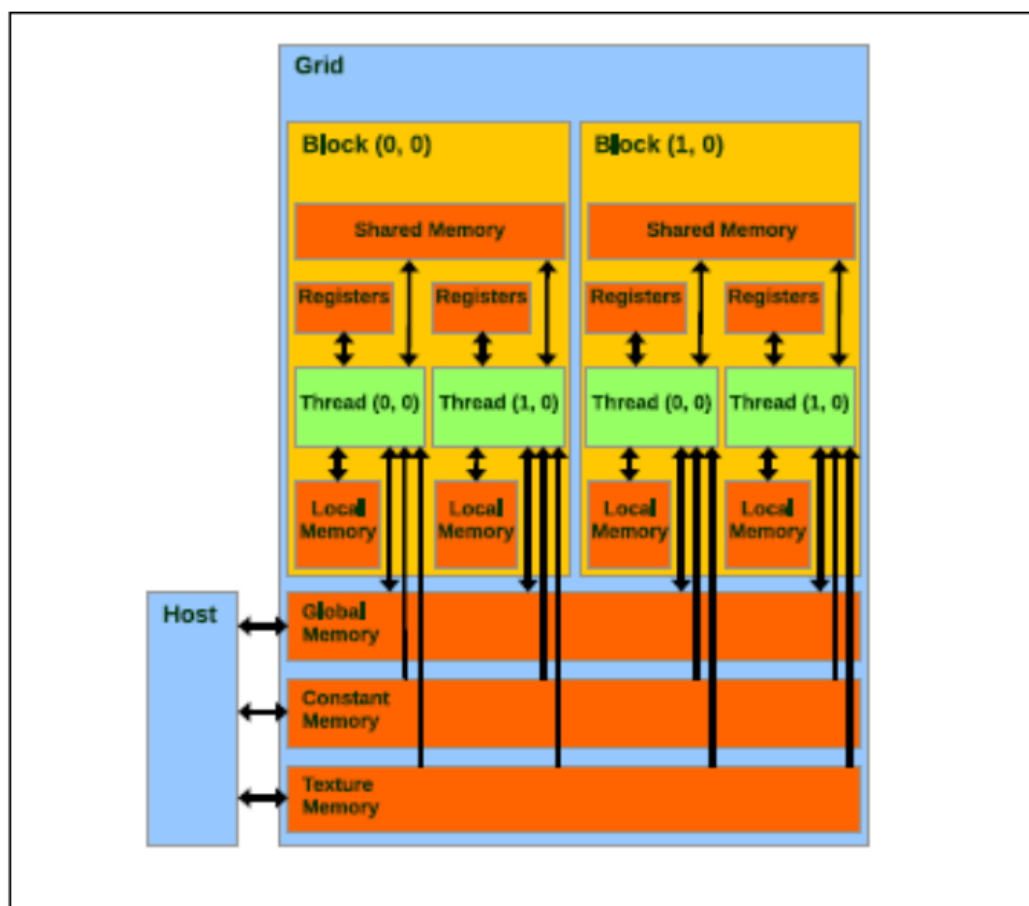


Figura 3. Visão mais detalhada dos componentes de memória da GPU. Fonte: [5].

O programador é responsável por definir a quantidade de *grids* e *threads* que cada *kernel* vai executar. Entretanto, ele não precisa estar ciente da quantidade física de *grids*, blocos e

processadores que a GPGPU possui, já que a própria interface do CUDA faz o gerenciamento para adequar a exigência do programa para as características de cada GPGPU.

A interface do CUDA permite que um programa escrito em CUDA seja compatível com todas as placas que dão suporte à tecnologia, desde a primeira placa com tecnologia CUDA lançada até as placas mais recentes. Funcionalidades mais recentes executadas nativamente em placas modernas também rodam em placas antigas, mas o CUDA faz uma emulação internamente.

Codificando CUDA

Um programa CUDA é um programa que é executado tanto na CPU (chamada de *host*) quanto na GPGPU (chamada de *device*). No programa, as funções que são executadas pelo *device* e invocadas pelo *host* possuem a anotação `__global__`. Podem ser escritas funções que são chamadas somente pelo *device*; essas funções possuem a anotação `__device__`. Por fim, funções que são executadas somente no *host* possuem a anotação `__host__`, entretanto essa anotação é opcional, visto que é a opção padrão no caso de nenhuma anotação.

Os *kernels*, funções que são executadas no *device*, não possuem retorno, justamente por serem executadas de forma assíncrona. Também, os *kernels* mandados para o *device* não devem ter resultados dependentes, visto que a ordem de execução não é garantida, sendo definida pela interface do CUDA de acordo com a situação.

São usadas funções especiais para fazer a manipulação de memória no *device*. Essas funções são:

- **CudaMalloc:** aloca memória.
 - `cudaError_t cudaMalloc(void **devPtr, size_t size);`
- **CudaFree:** libera a memória alocada na GPU.
 - `cudaError_t cudaFree(void *devPtr);`
- **CudaMemcpy:** copia dados entre a memória da CPU e a memória da GPU.
 - `cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind);`

É possível configurar as dimensões e os tamanhos de cada *grid* e bloco utilizados na computação de cada conjunto de *threads*. Essas informações são passadas como argumentos, estruturados da seguinte maneira quando um *kernel* é invocado pela CPU:

```
<<< dimensão e tamanho de cada grid, dimensão e tamanho de cada bloco >>>
```

A dimensão é definida pela natureza do processamento. O tipo aceito por essa estrutura é um tipo de dado especial chamado `dim3` que pode indicar até 3 dimensões.

Exemplos:

```
dim3 a(X); // cria uma variável de uma dimensão de tamanho X;
dim3 b(X, Y); // cria uma variável de duas dimensões com cada
dimensão com tamanho X e Y respectivamente;
dim3 c(X, Y, Z); // cria uma variável de três dimensões com cada
dimensão com tamanho X, Y e Z respectivamente;
```

Utilizar um inteiro tem o mesmo efeito de utilizar um `dim3` inicializado com somente uma dimensão.

Executando CUDA

Os códigos de exemplo mostrados aqui são códigos CUDA escritos em C, mas CUDA possui interfaces para diversas outras linguagens como C++, C#, Fortran, Java, Python etc.

O código a seguir, disponível também no arquivo [dobra-valores.cu](https://github.com/valores/cuda-dobra-valores), traduzido de [7], mostra um exemplo de programa CUDA mínimo que dobra cada elemento de um *array* de tamanho *N* utilizando a GPU.

```
#include <stdio.h>

// Exemplo quase mínimo de CUDA. Compile com:
// nvcc -o dobra-valores dobra-valores.cu

#define N 1000

// A função marcada com __global__ roda na GPU mas pode ser
```

```

chamada pela CPU
// Essa função multiplica os elementos
// de um array de inteiros por 2
// Toda computação pode ser pensada como rodar com
// uma thread por elemento do array com blockIdx.x
// identificando a thread
//
// A comparação i<N ocorre porque não é conveniente ter
// exatamente uma correspondência de 1 para 1 entre threads
// e elementos do array. Não é realmente necessário aqui
//
// Note como estamos misturando código de GPU e CPU no
// mesmo arquivo fonte. Uma forma alternativa de usar CUDA
// é manter o código C/C++ separado do código cuda
// compilar e carregar esse código dinamicamente em
// tempo de execução, um pouco semelhante a como shaders
// são compilados e carregados pelo código C/C++ no OpenGL
__global__
void add(int *a, int *b) {
    int i = blockIdx.x;
    if (i<N) {
        b[i] = 2*a[i];
    }
}

int main() {
    // Cria um array na CPU.
    // ('h' indica host, por convenção, não é obrigatório)
    int ha[N], hb[N];

    // Cria um array correspondente na GPU.
    // ('d' indica device, por convenção, não é obrigatório)
    int *da, *db;
    cudaMalloc((void **)&da, N*sizeof(int));
    cudaMalloc((void **)&db, N*sizeof(int));

```

```

// Inicializa os dados na CPU.
for (int i = 0; i<N; ++i) {
    ha[i] = i;
}

// Copia os dados para o array da GPU.
cudaMemcpy(da, ha, N*sizeof(int), cudaMemcpyHostToDevice);

// Inicia o código em GPU com N threads, uma por elemento do
array.
add<<<N, 1>>>>(da, db);

// Copia o resultado do array da GPU para o da CPU.
cudaMemcpy(hb, db, N*sizeof(int), cudaMemcpyDeviceToHost);

for (int i = 0; i<N; ++i) {
    printf("%d\n", hb[i]);
}

// Libera o espaço na GPU.
cudaFree(da);
cudaFree(db);

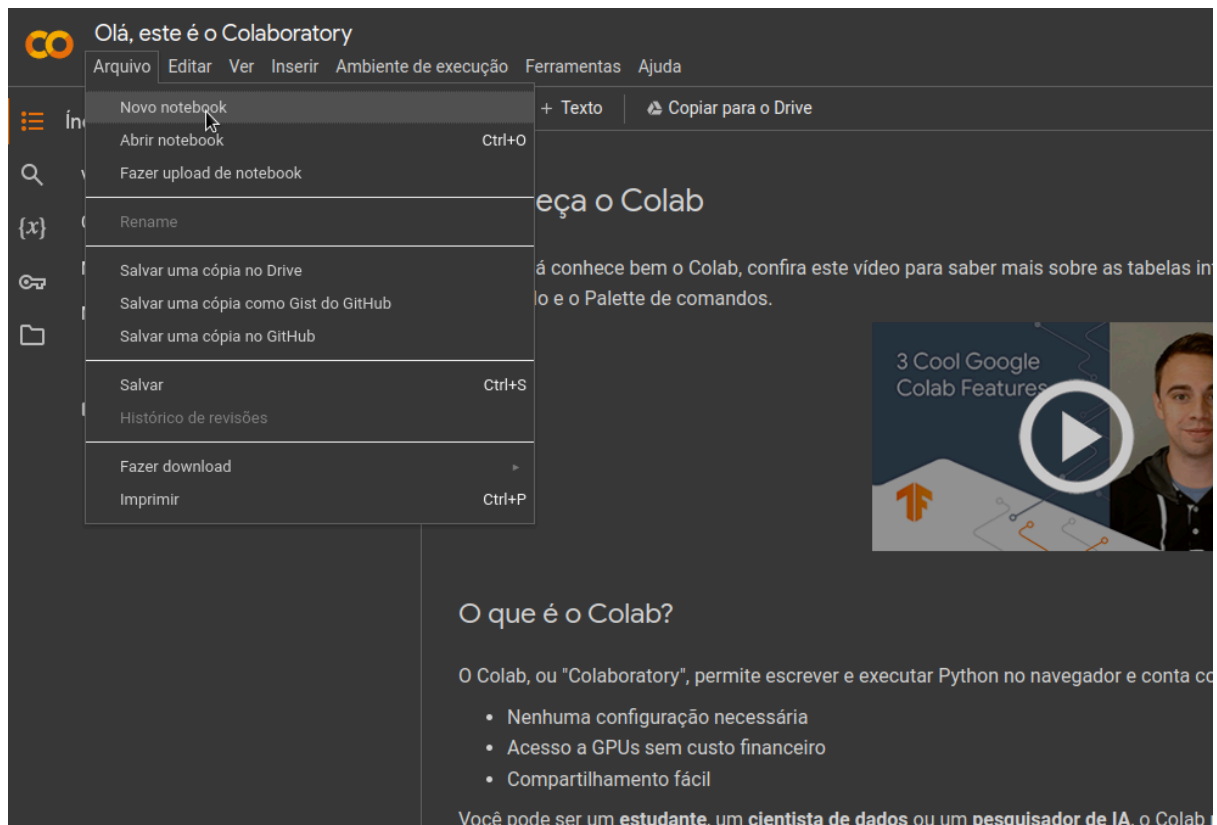
return 0;
}

```

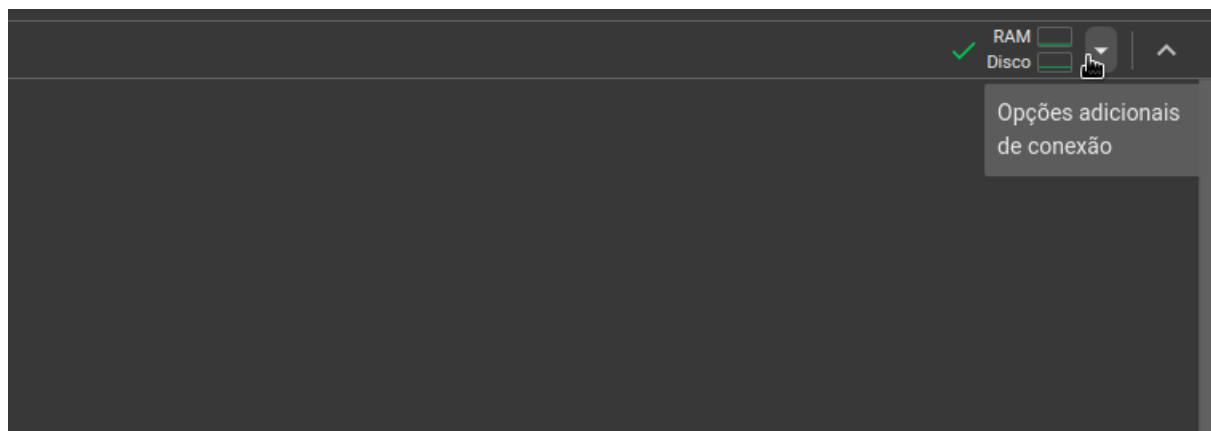
Programas CUDA só rodam em GPUs da NVIDIA, por isso será mostrado passo a passo como executar programas CUDA em GPUs na nuvem, utilizando a plataforma Google Colab². Para acessar o Google Colab é necessário ter uma conta Google.

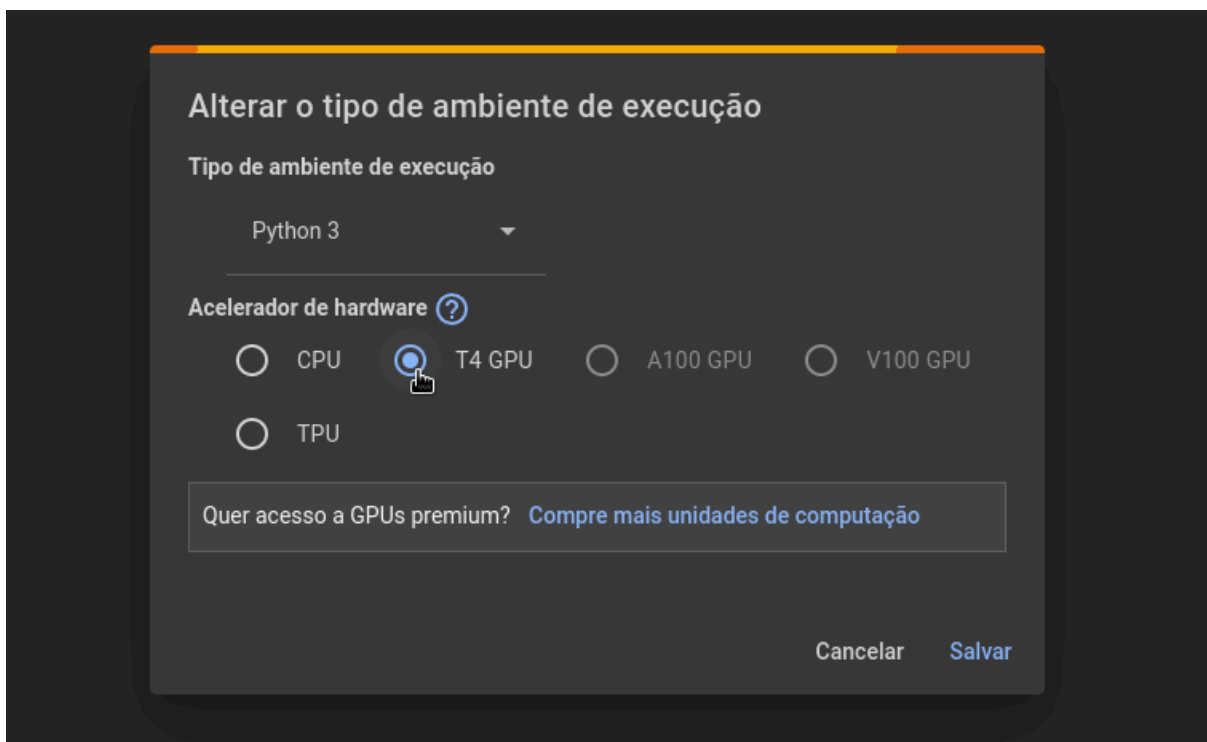
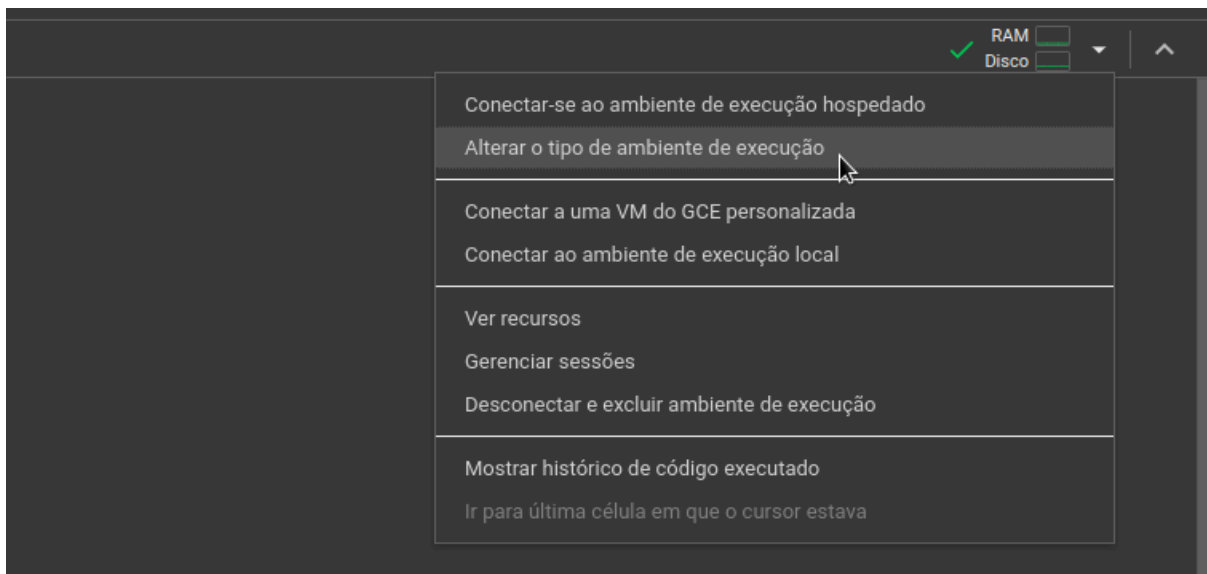
- 1) No Colab, crie um novo Notebook (*Arquivo* → *Novo notebook*):

² <https://colab.research.google.com/>

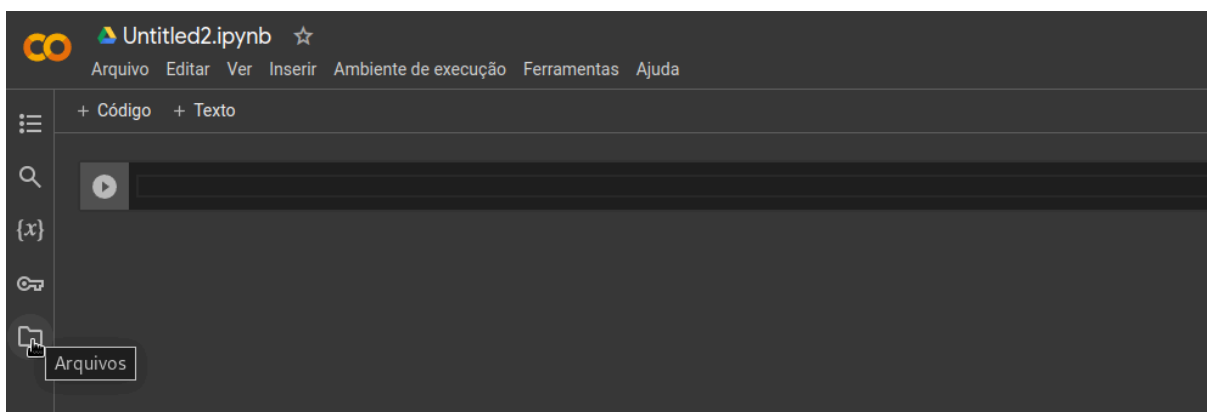


2) Altere o tipo de ambiente para GPU:

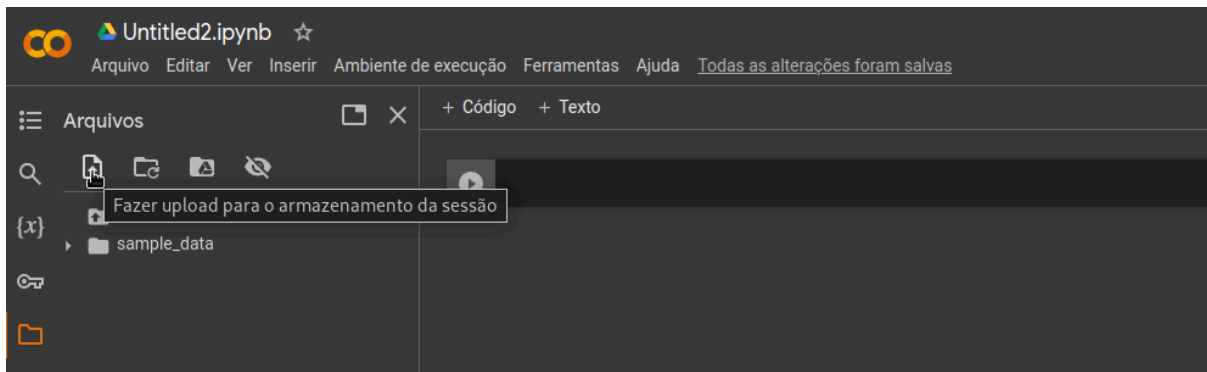




3) Clique em “Arquivos”:



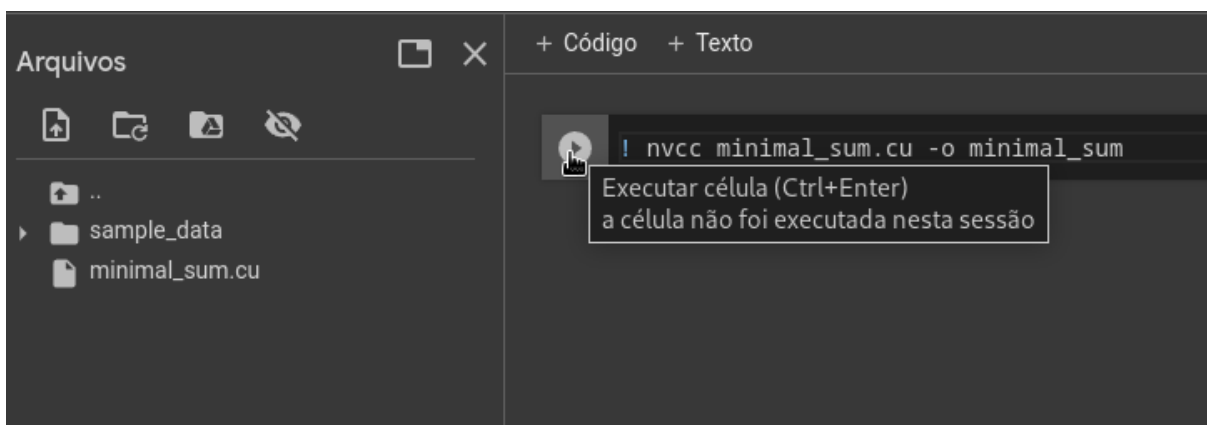
4) Selecione a opção “Fazer upload para o armazenamento da sessão”:



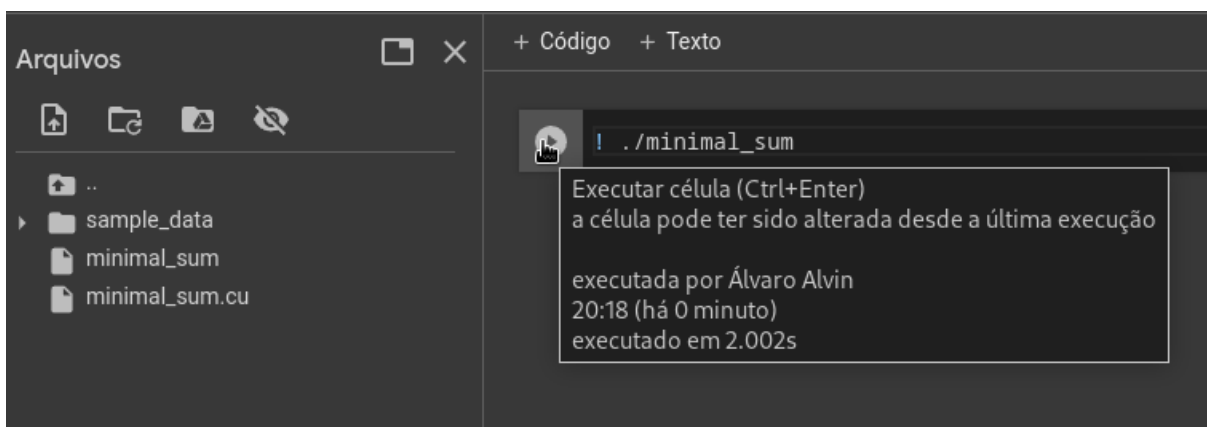
5) Selecione o arquivo `dobra-valores.cu`.

6) Agora é possível fazer a compilação do programa. Para isso, é possível executar comandos com o padrão do *shell* do Linux iniciando por '!'. A figura a seguir mostra como é possível fazer a compilação de um arquivo `.cu` utilizando o compilador `nvcc`. Para compilar o arquivo `dobra-valores.cu`, é necessário usar o comando:

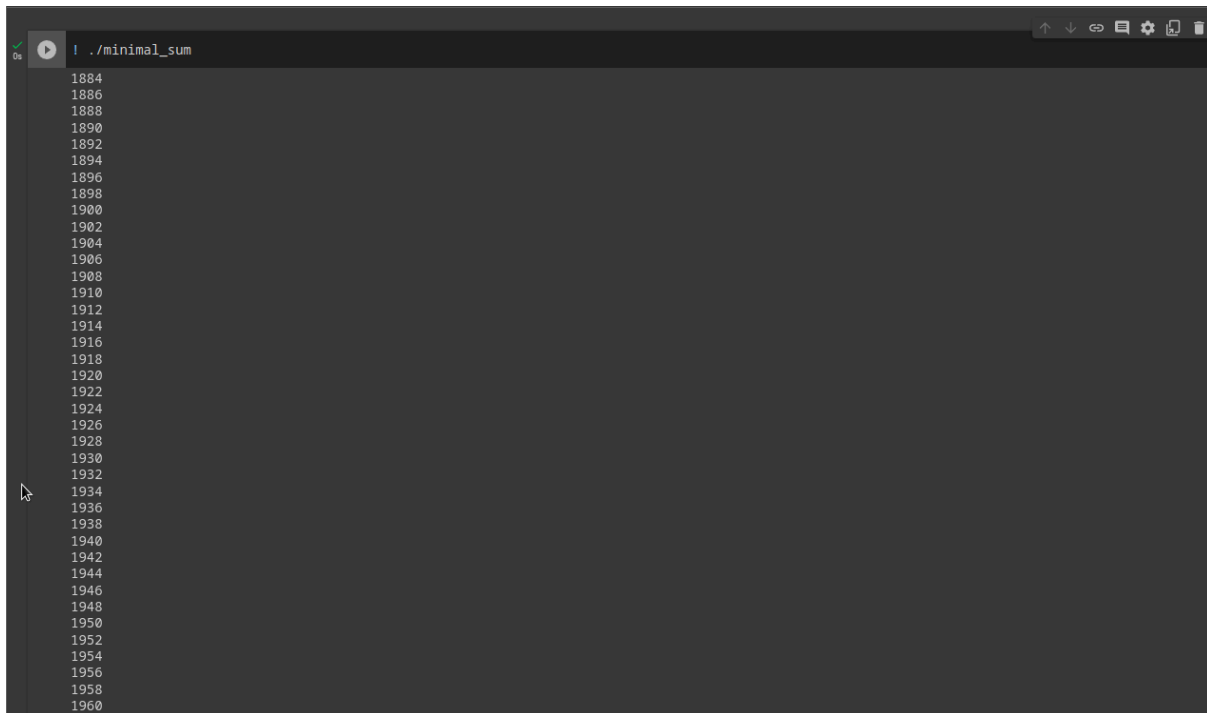
```
! nvcc dobra-valores.cu -o dobra-valores
```



7) E para executar o código, usa-se o comando `! ./dobra-valores`, semelhante ao que é feito na figura a seguir com um arquivo de teste chamado `minimal_sum`.



8) A saída exibida é:



A screenshot of a terminal window with a dark background. The title bar at the top shows a green checkmark icon, a play button icon, and the text `./minimal_sum`. The terminal displays a vertical list of even numbers starting from 1884 and ending at 1960, with an increment of 2 between each number. The numbers are aligned to the left of the terminal window.

O código a seguir, disponível no arquivo [soma-elementos-arrays.cu](https://github.com/rogerdmo/soma-elementos-arrays.cu) (adaptado de [8]), exemplifica o uso da função de endereçamento de memória unificado, fazendo uma única alocação de dados para o uso da CPU e da GPU:

```
#include "stdio.h"
#include "math.h"
// Kernel para somar dois elementos de um array
__global__ void add(int n, float *x, float *y) {
    /*
    como a quantidade de threads e blocos é personalizada,
    cada thread precisa saber exatamente qual parte do array
    deve pegar para fazer a computação
    */
    // o índice é dado pelo índice do bloco * dimensão do bloco *
    índice da thread
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    // o incremento é dado pela dimensão do bloco * dimensão do
    grid
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```

```

}

int main(void) {
    int N = 1 << 20;
    float *x, *y;

    // Aloca memória unificada, para uso da CPU e GPU
    cudaMallocManaged(&x, N * sizeof(float));
    cudaMallocManaged(&y, N * sizeof(float));

    // inicializa os arrays no host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Roda o kernel em 1M elementos na GPU
    // define quantas threads por blocos
    int blockSize = 256;
    // define quantos blocos são necessários para 1M de elementos
    int numBlocks = (N + blockSize - 1) / blockSize;
    // executa na GPU definindo a quantidade de blocos e threads
    (por bloco) utilizadas
    add<<<numBlocks, blockSize>>>(N, x, y);

    // Aguarda a GPU terminar a computação
    cudaDeviceSynchronize();



    // Exibe os valores calculados, todos devem ser 3.0
    for (int i = 0; i < N; i++) {
        printf("%f\n", y[i]);
    }

    // Libera a memória alocada
    cudaFree(x);
    cudaFree(y);
}

```

```
    return 0;  
}
```

Vídeos Sugeridos

1.  [CUDA Hardware](#)
2.  [CUDA Programming](#)

Observações

- Por vezes, a plataforma Google Colab pode apresentar instabilidade. Essas situações são imprevisíveis, não havendo o que o usuário possa fazer para contorná-las.
- Pode ser que o Google Colab exija uma configuração de localidade (*locale*) UTF-8 para executar comandos. Um erro pode ocorrer se a configuração estiver definida como ANSI_X3.4-1968, pois esta não suporta caracteres UTF-8. Para contornar este problema, muito comum ao digitar comandos como `!ls -l` e `!nvidia-smi -q`, é necessário alterar a codificação de caracteres para UTF-8. Nesse sentido, sugere-se digitar os comandos abaixo no início do *notebook Colab*:

```
import locale  
locale.getpreferredencoding = lambda: "UTF-8"
```

Resumo

Neste documento foi apresentada brevemente a estrutura de uma GPU e suas características que por si só fazem com a GPU tenha uma grande poder computacional para realizar um tipo de processamento não só gráfico com múltiplos dados sendo estes independentes entre si, então foi apresentada a interface que permite desenvolver programas que utilizam essas características em placas da Nvidia, o CUDA, e por fim foi apresentado conceitos básicos sobre a implementação desses programa, exemplos, e uma maneira de executá-los em GPUs na nuvem.

Exercícios

- 1) A arquitetura da GPU é ideal para qual tipo de operação? Assinale a alternativa correta:
 - a) Múltiplas instruções com múltiplos dados;
 - b) Multiplus instruções com um dado;
 - c) Única instrução com múltiplos dados;
 - d) Única instrução com único dado.

- 2) Sobre GPUs, arquitetura e programas CUDA, analise as afirmações.
 - I. Programas CUDA podem ser executados em qualquer GPU.
 - II. A interface CUDA é responsável por organizar a execução das operações na GPU, mesmo que os parâmetros passados para execução do *kernel*, como número de *threads* e blocos não sejam compatíveis com a GPU do sistema.
 - III. Existem funções oferecidas pela interface CUDA que só podem ser certas GPUs dentro do conjunto de GPUs que são compatíveis com CUDA.
 - IV. Cada *thread* tem acesso a três memória diferentes, uma exclusiva dela, outra compartilhada entre as *threads* de um mesmo bloco e ou global, compartilhada por toda a GPU.

Quais estão corretas?

 - a) I, II e IV.
 - b) II, III e IV.
 - c) II e IV.
 - d) Somente IV.

- 3) Modifique o primeiro programa para utilizar a memória unificada e utilizar mais de uma *thread* para o processamento dos *kernels*. Quando mais de uma *thread* por bloco é utilizada, é preciso calcular a posição do dado que cada *kernel* precisa acessar de acordo com seu bloco e identificador.

Gabarito

- 1) c
- 2) c
- 3)

```

#include <stdio.h>
#define N 1000

__global__ void add(int *a, int *b)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < N; i += stride)
        b[i] = 2 * a[i];
}

int main() {
    int *a, *b;
    cudaMallocManaged(&a, N * sizeof(float));
    cudaMallocManaged(&b, N * sizeof(float));

    for (int i = 0; i < N; ++i)
        a[i] = i;

    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    add <<< blockSize, numBlocks >>> (a, b);

    cudaDeviceSynchronize();

    for (int i = 0; i < N; ++i)
        printf("%d\n", b[i]);

    cudaFree(a);
    cudaFree(b);

    return 0;
}

```


Referências

[1] AMD. AMD Introduces New AMD Ryzen Threadripper 7000 Series Processors and Ryzen Threadripper PRO 7000 WX-Series Processors for the Ultimate Workstation.

Disponível em:

<https://www.amd.com/en/newsroom/press-releases/2023-10-19-amd-introduces-new-amd-ryzen-threadripper-7000-ser.html>. Acesso em: 06 ago. 2024.

[2] AMD. 4th Generation AMD EPYC™ Processors. Disponível em:

<https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html>. Acesso em: 06 ago. 2024.

[3] NVIDIA. GeForce RTX 4090. Disponível em:

<https://www.nvidia.com/pt-br/geforce/graphics-cards/40-series/rtx-4090/>. Acesso em: 06 ago. 2024.

[4] GHORPADE, J. GPGPU Processing in CUDA Architecture. Advanced Computing: An International Journal, v. 3, n. 1, p. 105–120, 31 jan. 2012.

[5] LIPPERT, A. NVIDIA GPU Architecture for General Purpose Computing. [s.l.: s.n.].

Disponível em: <https://www.cs.wm.edu/~kemper/cs654/slides/nvidia.pdf>. Acesso em: 06 ago. 2024.

[6] OH, Fred. What Is CUDA? NVIDIA Blog. Disponível em:

<https://blogs.nvidia.com/blog/what-is-cuda-2/>. Acesso em: 06 ago. 2024.

[7]. PIPONI, Dan. `example.cu`. Github. Disponível em:

<https://gist.github.com/dpiponi/1502434>. Acesso em: 06 ago. 2024.

[8]. HARRIS, Mark. An Even Easier Introduction to CUDA. Disponível em:

<https://developer.nvidia.com/blog/even-easier-introduction-cuda/>. Acesso em: 06 ago. 2024.