

Introdução à Programação de GPUs com CUDA

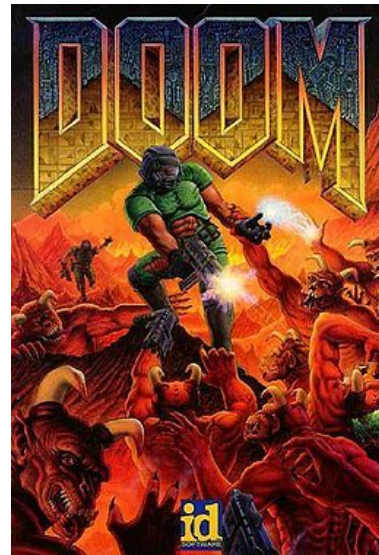
Lucas de Sousa Rosa e Alfredo Goldman
MAC0219 - Programação Concorrente e Paralela

23 de outubro de 2024

Motivação

O que Doom pode nos ensinar sobre GPUs?

"It Runs Doom" é uma expressão que indica que um dispositivo é capaz de rodar **Doom 1993**, um jogo de tiro em primeira pessoa, que foi portado com sucesso para uma variedade de dispositivos eletrônicos projetados para outros propósitos além de jogos.



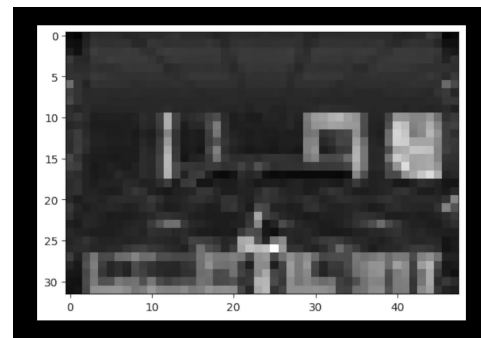
O que Doom pode nos ensinar sobre GPUs?



Osciloscópio (2006)



Teste de Gravidez (2020)



E. Coli (2023)

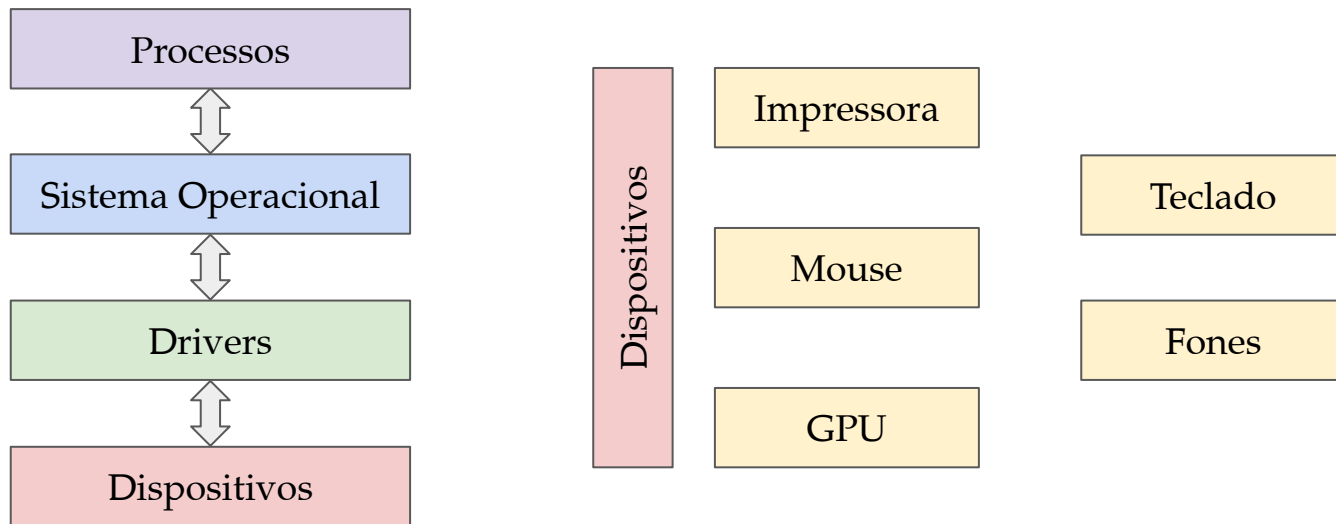
Veja também: <https://canitrundoom.org/>

O que Doom pode nos ensinar sobre GPUs?

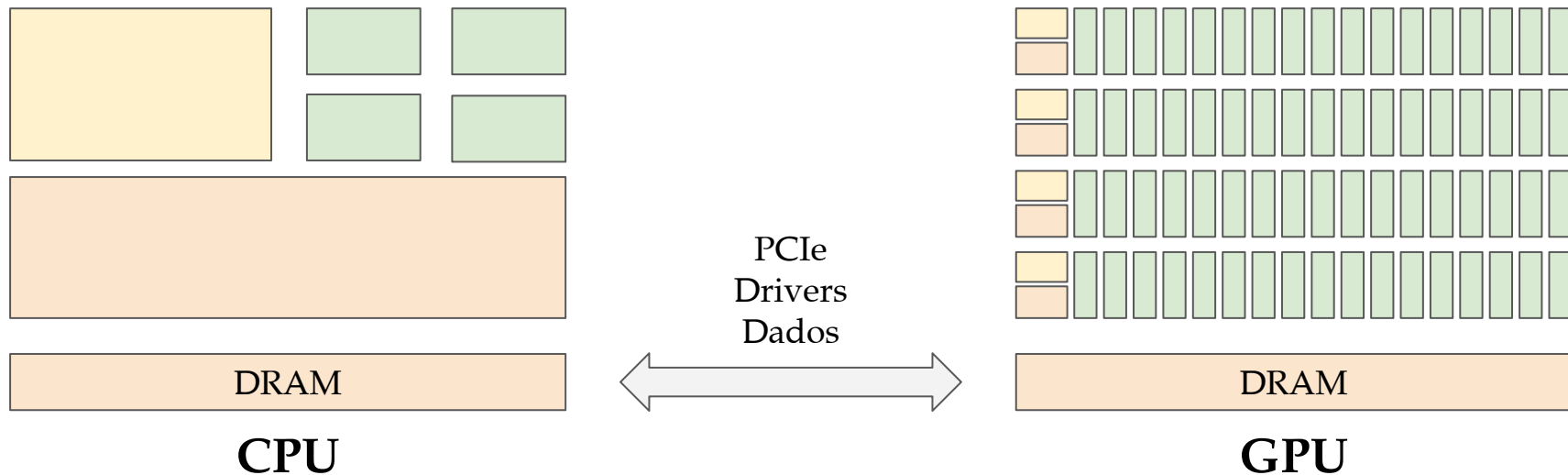
<https://github.com/jhuber6/doomgeneric>

Um port de Doom que roda (quase) inteiramente na GPU usando a libc de LLVM para GPUs e baseado na interface doomgeneric.

O que isso tem de interessante? GPUs já não são usadas para processar gráficos?



O que Doom pode nos ensinar sobre GPUs?



Os processos precisam "passar" pelo SO para qualquer E/S.

A linguagem C abstrai essa "passagem" através de chamadas de sistema (*syscalls*).

O projeto LLVM implementa chamadas remotas a partir da GPU para o sistema operacional.

O que Doom pode nos ensinar sobre GPUs?

```
37     int DG_GetKey(int *pressed, unsigned char *doomKey) {
38         rpc_host_call(get_input, &key_buffer, sizeof(uint32_t *));
39         if (*key_buffer == 0)
40             return 0;
41
42         *pressed = *key_buffer >> 8;
43         *doomKey = *key_buffer & 0xFF;
44
45         return 1;
46     }
```

Exemplo de chamada de sistema do port de Doom.

O que Doom pode nos ensinar sobre GPUs?

Por que alguém faria isso?

why

Because I can.

A resposta segundo o autor do port.

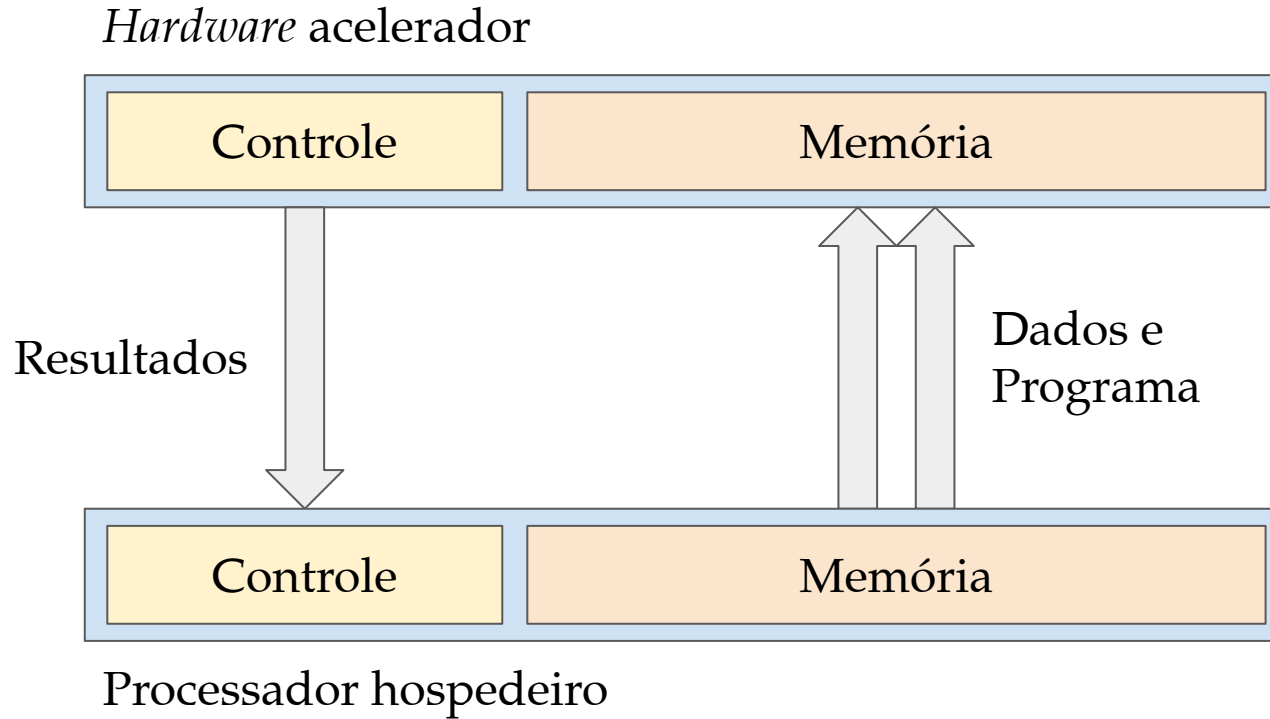
Computação Heterogênea

Aceleração por *hardware*

Uso de **dispositivos** (*devices*) para acelerar computações aplicadas a conjuntos de dados (geralmente grandes).

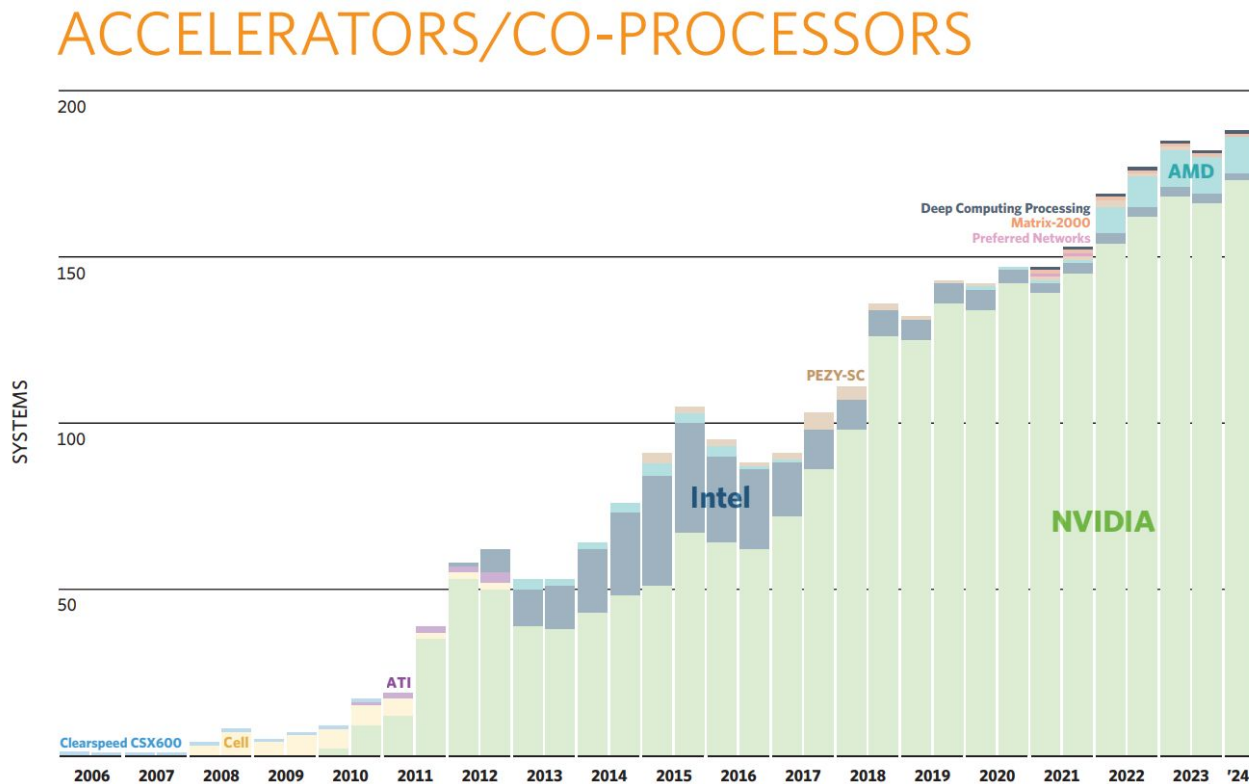
- Associação a um processador hospedeiro (*host*).
- **Controle e memória** próprios.
- Diferem em especialização e configurabilidade.
- GPUs, DSPs, FPGAs, ASICs.

Aceleração por *hardware*

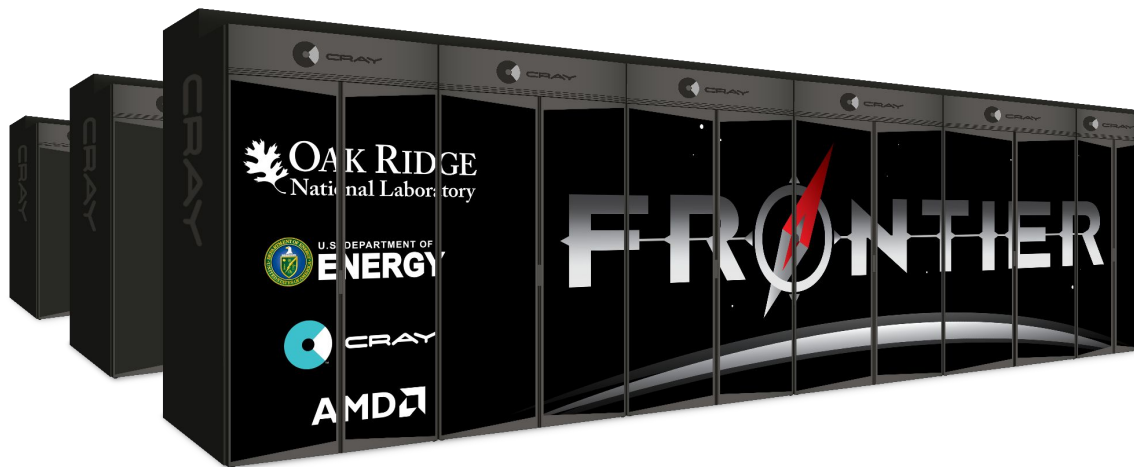


Aceleração por *hardware*

Evolução da presença
de aceleradores no
TOP500.



Computação heterogênea



Dados recursos computacionais **heterogêneos**, conjuntos de **dados** e **computações**, como distribuir computações e dados de forma a **otimizar o uso** dos recursos?

Computação heterogênea

Recursos heterogêneos

- Baixa latência: CPUs.
- Alta vazão (*throughput*): GPUs.
- Especializados: ASICs, DSPs.
- Reconfiguráveis: FPGAs.
- Memória, disco, etc.

Conjunto de dados

- *Big Data*.
- *Data Streams*.
- ...

Modelos de programação

- *MapReduce*.
- *Task Parallelism*.
- ...

GPUs

Graphics Processing Units (GPUs)

Originalmente especializadas em **processamento gráfico**, trabalham com muitos dados e têm **alta vazão**.

- Sem *branch prediction*.
- Milhares de ALU de **maior latência**.
- *Pipelines* de execução.
- Uma H100 comporta até 270.336 threads concorrentes.



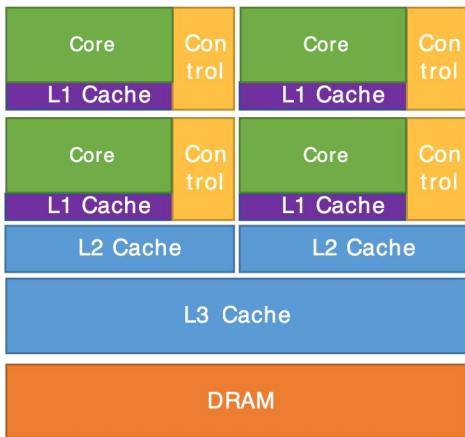
GPUs versus CPUs

CPUs

Processamento de instruções mais sofisticado

Velocidade de clock mais rápida

Baixa latência (*cache*)



CPU

GPUs

Muitas unidades de processamento

Maior largura de banda

Planejada para workloads paralelos



GPU

Redefinindo alguns conceitos

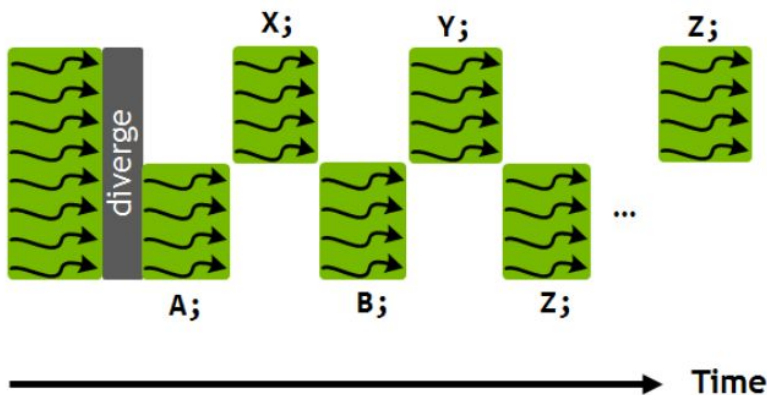
Terminologia (GPU)	Definição	Equivalente (CPU)
<i>thread</i>	O fluxo de instruções e dados que é atribuído a um CUDA <i>core</i> ; Single Instruction, Multiple Threads (SIMT)	N/A
CUDA <i>core</i>	Unidade que processa um item de dados após o outro; executa parte de um fluxo de instruções SIMT	<i>vector lane</i>
<i>warp</i>	Grupo de 32 <i>threads</i> que executam o mesmo fluxo de instruções em dados distintos	<i>vector</i>
<i>kernel</i>	Função que roda no dispositivo; um <i>kernel</i> pode ser subdividido em blocos de threads	<i>thread(s)</i>
SM, <i>streaming multiprocessor</i>	Unidade capaz de executar um bloco de <i>thread</i> de um <i>kernel</i> ; vários SMs podem trabalhar juntos em um <i>kernel</i>	<i>core</i>

SIMT e warps

Single Instruction, Multiple Data (SIMD): uma instrução age da mesma forma em diferentes dados.

Single Instruction, Multiple Threads (SIMT): execução menos restrita. *Threads* podem ser ativadas e desativadas. O contexto das *threads* ativadas continua inalterado (podendo ser recuperado).

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



SIMT e warps

Um bloco de *threads* é dividido em *warps* para execuções SIMT.

Um *warp* completo é composto por um grupo de 32 *threads* consecutivas.

As *threads* de um *warp* são processadas por 32 CUDA *cores*.

Cada grupo é análogo às unidades de processamento vetorial de CPUs.

Kernels e SMs

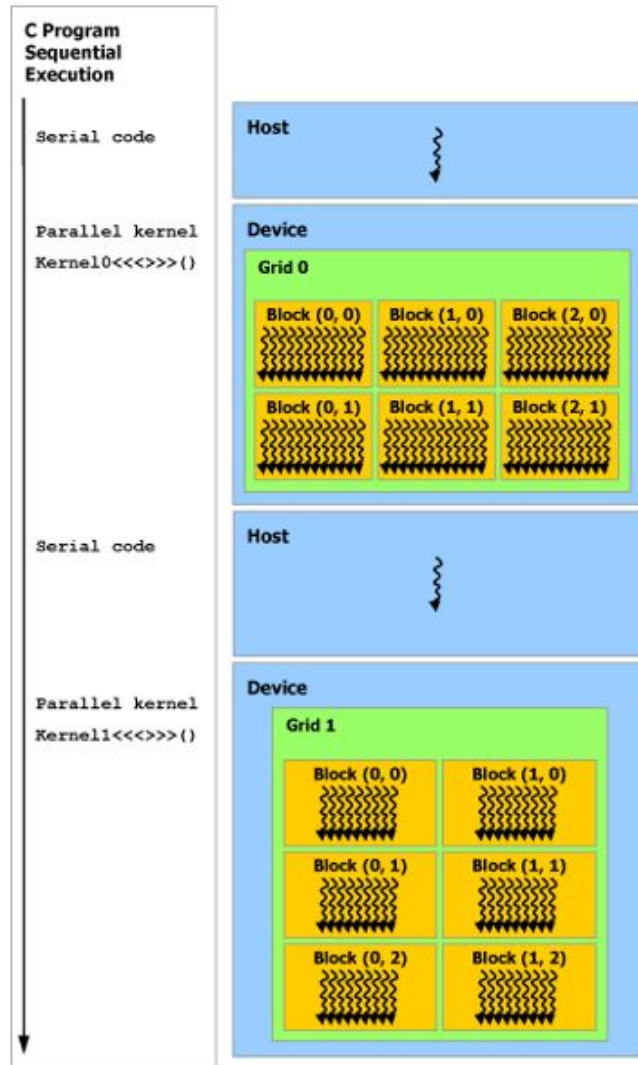
Kernels são funções a serem executadas em paralelo.

Kernels podem ser executados em N vezes por N *threads* diferentes.

Kernels podem ser executados por um ou mais *streaming multiprocessors*, SMs.

As SMs armazenam os CUDA *cores*; além de criar, escalonar e executar *warps*.

As SMs também possuem hierarquias de memória como registradores, *cache* L1, *caches* constantes e memória compartilhada.



Compute capability

É o conjunto de características disponíveis que diferenciam as arquiteturas.

- *Compute capability 7.x* = arquitetura Volta
- *Compute capability 7.5* = arquitetura Turing
- *Compute capability 8.x* = arquitetura Ampere



NVIDIA Tesla V100, baseada na arquitetura Volta, e NVIDIA Quadro RTX 5000, baseada na arquitetura Turing.

Destrinchando uma NVIDIA Tesla V100



NVIDIA Tesla V100 (2017)

Arquitetura Tesla

16 GB HBM2

Especificações

- 2560 FP64 CUDA *cores*.
- 5120 FP32 CUDA *cores*.
- Suporte a *fused multiply-add* (FMA).
- 1,53 GHz de frequência.

Pico de desempenho

- 7,8 teraflop/s (*double precision*).
- 15,7 teraflop/s (*single precision*).

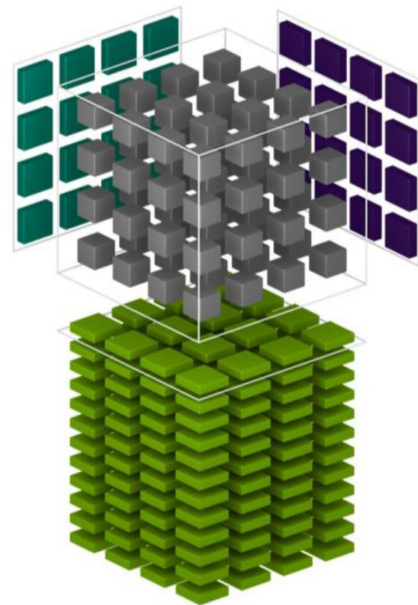
Destrinchando uma NVIDIA Tesla V100



Destrinchando uma NVIDIA Tesla V100

Organização de um SM

- Tipos de CUDA *cores*:
 - 64 FP32 CUDA *cores*
 - 64 INT32 CUDA *cores*
 - 32 FP64 CUDA *cores*
- 8 *Tensor Cores*
- 16 *Special Function Units*
- 4 *Texture units*



CUDA & CUDA C

Aceleração por *software*

Formas de interagir com a GPU: bibliotecas, diretivas de compilação e linguagens de programação.

- **Bibliotecas** são fáceis de usar (otimizada por especialistas).
- **Diretivas de compilação** são tão fáceis quanto OpenMP (desempenho depende do compilador).
- **Linguagens de programação** são mais difíceis de usar (desempenho depende da sua competência).
 - Vamos nos concentrar aqui.

Para ter acesso as ferramentas do CUDA é preciso instalar o CUDA *toolkit*.



cuBLAS

GPU-accelerated basic linear algebra (BLAS) library.

[Learn More >](#)



cuFFT

GPU-accelerated library for Fast Fourier Transform implementations.

[Learn More >](#)



cuRAND

GPU-accelerated random number generation.

[Learn More >](#)



cuSOLVER

GPU-accelerated dense and sparse direct solvers.

[Learn More >](#)



cuSPARSE

GPU-accelerated BLAS for sparse matrices.

[Learn More >](#)



cuTENSOR

GPU-accelerated tensor linear algebra library.

[Learn More >](#)



cuDSS

GPU-accelerated direct sparse solver library.

[Learn More >](#)



CUDA Math API

GPU-accelerated standard mathematical function APIs.

[Learn More >](#)



AmgX

GPU-accelerated linear solvers for simulations and implicit unstructured methods.

[Learn More >](#)

CUDA C



C	CUDA
<pre>void c_hello(){ printf("Hello World!\n"); } int main() { c_hello(); return 0; }</pre>	<pre>__global__ void cuda_hello(){ printf("Hello World from GPU!\n"); } int main() { cuda_hello<<<1,1>>>(); return 0; }</pre>

- A palavra-chave `__global__` identifica uma função como *kernel*.
 - *Kernels* devem ter o tipo `void`.
- A sintaxe `<<<...>>>` serve para lançar e configurar um *kernel*.
 - Veremos os detalhes mais a frente.
- Arquivos CUDA devem ter extensão `.cu` e podem ser compilados com o compilador `nvcc` (inclusive no CUDA *toolkit*).

Soma de vetores (CUDA)

- Versão de CPU: `1_vector_add.c`.
- Versão de GPU (errada): `2_vector_add.cu`.
- Por que o código não funciona?
 - Os dados não estão na memória da GPU!!

O que fazer?

1. Alocar memória do *host* e inicializar os dados do *host*. 
2. Alocar memória do *device*.
3. Transferir dados de entrada da memória do *host* para a memória do *device*.
4. Executar *kernels*. 
5. Transferir saída da memória do *device* para o *host*.

Soma de vetores (CUDA)

Gerenciamento de memória

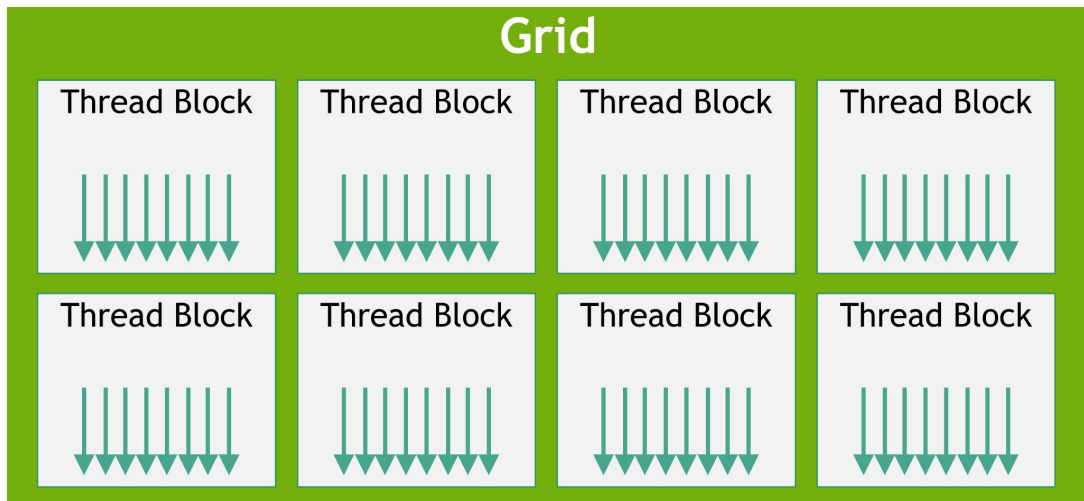
- `cudaMalloc(void **devPtr, size_t count);`
- `cudaFree(void *devPtr);`
- `cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind)`

É preciso importar as bibliotecas `cuda.h` e `cuda_runtime.h`.

- **Versão de GPU (correta):** `3_vector_add.cu`.
 - Mensurando o tempo de execução...
- O comando `time` não nos dá muita informação relevante.
 - `nvprof`

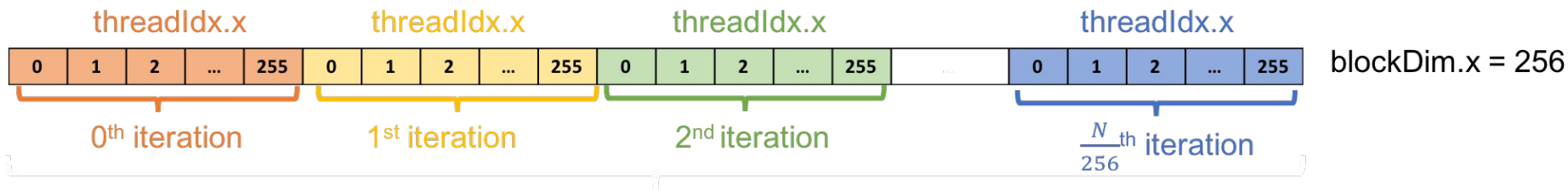
Soma de vetores (CUDA)

- Até agora não exploramos a configuração do lançamento do *kernel*.
 - `<<<...>>>`
- CUDA organiza *threads* em blocos de *threads*. Um *kernel* pode lançar múltiplos blocos organizados em uma estrutura de *grid*.
 - `<<<M, T>>>` significa “lance um *grid* de M blocos de *thread* e T *threads* por bloco”.



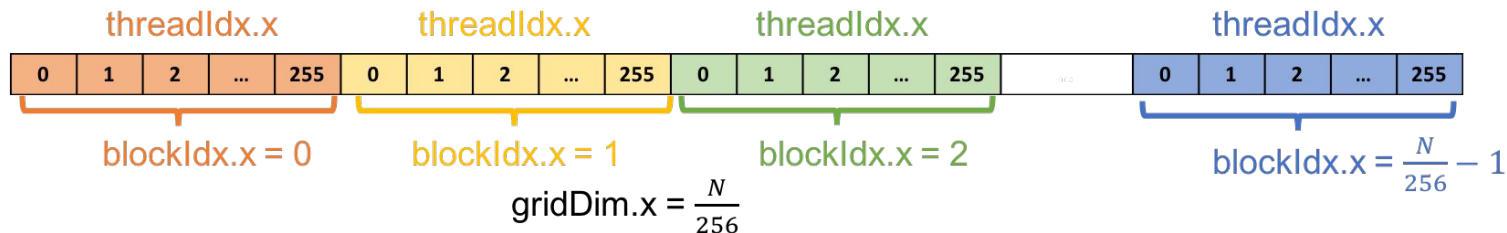
Soma de vetores (CUDA)

- Vamos adicionar mais *threads*!!
 - Vamos considerar **256 threads** por bloco: `<<<1, 256>>>`
 - Ainda precisamos indicar a operação que cada *thread* vai fazer.
- `threadIdx.x`: índice da *thread* no bloco.
- `blockDim.x`: tamanho do bloco (número de *threads* no bloco).
- Implementação de 1 bloco: `3_vector_add.cu`.



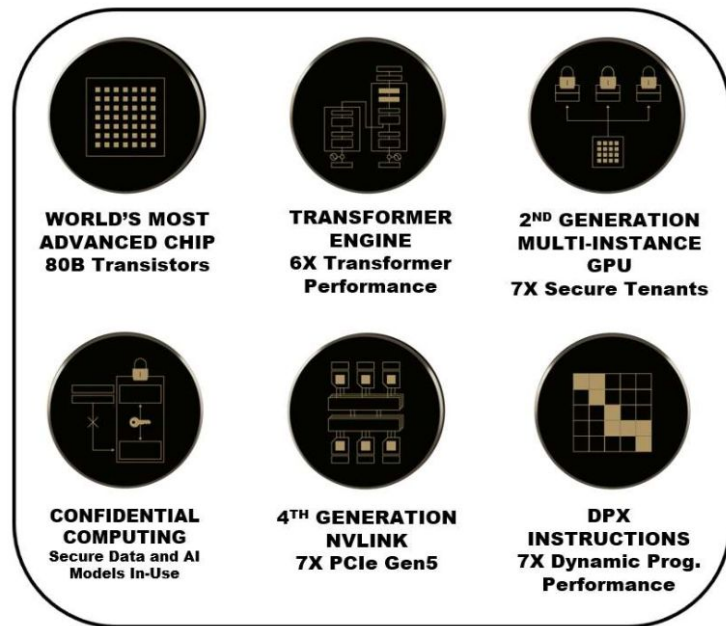
Soma de vetores (CUDA)

- Vamos aproveitar ainda mais a GPU e executar múltiplos blocos de *threads*.
- Lembrando que cada SM pode executar vários blocos de *threads* ao mesmo tempo.
 - Os SMs executam as *threads* em grupos de 32 (um *warp*).
 - Blocos de 256 *threads* equivalem a 8 *warps*.
- `blockIdx.x`: índice do bloco no *grid*.
- `gridDim.x`: tamanho do *grid* (número de blocos no *grid*).
- Implementação de múltiplos blocos: `5_vector_add.cu`.



Soma de vetores (CUDA)

- Escrever código CUDA pode ser muito complexo e difícil.
 - **Dica prática:** sempre procure por erros!
 - **Implementação da NVIDIA:**
`6_vector_add.cu`.
- Só vimos um pedacinho muito pequeno do que as GPUs são capazes.



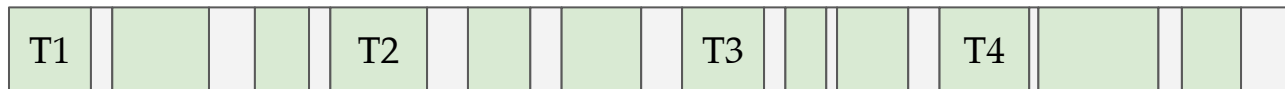
Tecnologias da arquitetura Hopper (H100)

Referências

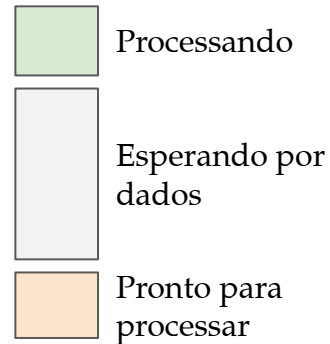
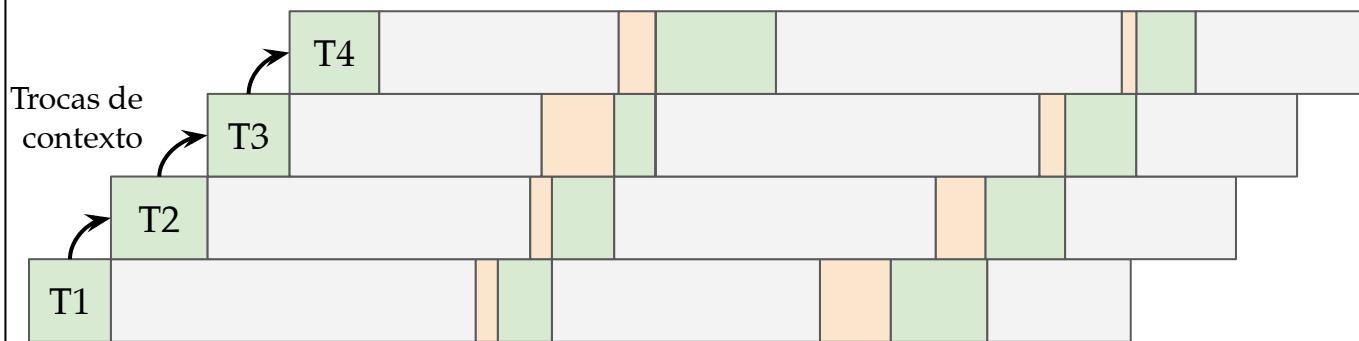
- Os códigos-fonte utilizados e outras aulas estão disponíveis no repositório do Github **fredgrub/aulas-PPD**.
- Slides do Pedro e mais exemplos: <https://github.com/phrb/intro-cuda>
- [Doom rodando na GPU: como? \(portaram um SO pra GPU só pra isso\).](#)
- <https://cvw.cac.cornell.edu/gpu-architecture>
- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
- <https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/>
- <https://nyu-cds.github.io/python-gpu/02-cuda/>

GPU versus CPU

CPU:



GPU:



Threads de GPU são bastante simples e leves. Trocas de contexto quase não tem custo.