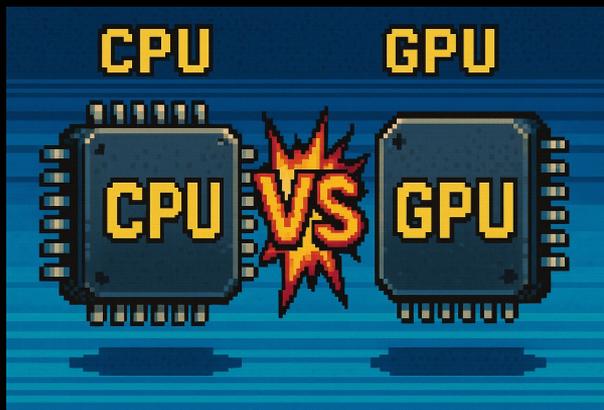




BY

“Às vezes, grande e burro é melhor” (Patterson, Henessy - Quantitative Approach; 2019).



Paulo Lisboa de Almeida



# CPU

Voltando ao básico.

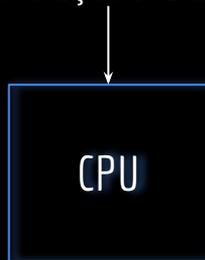
CPU de apenas “um núcleo”, que processa as instruções uma a uma.

Cada instrução opera em um dado.

SISD - Single Instruction, Single Data.

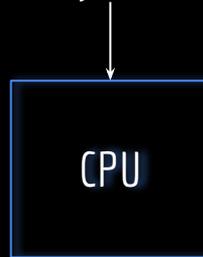
# SISD

...  
Instrução4 Dado4  
Instrução3 Dado3  
Instrução2 Dado2  
Instrução1 Dado1

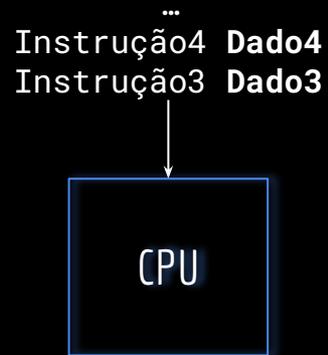


# SISD

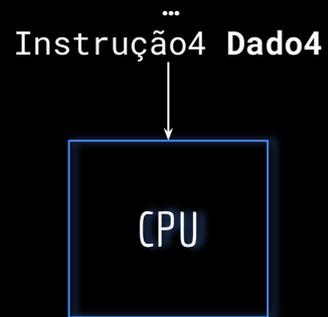
...  
Instrução4 Dado4  
Instrução3 Dado3  
Instrução2 Dado2



# SISD



# SISD



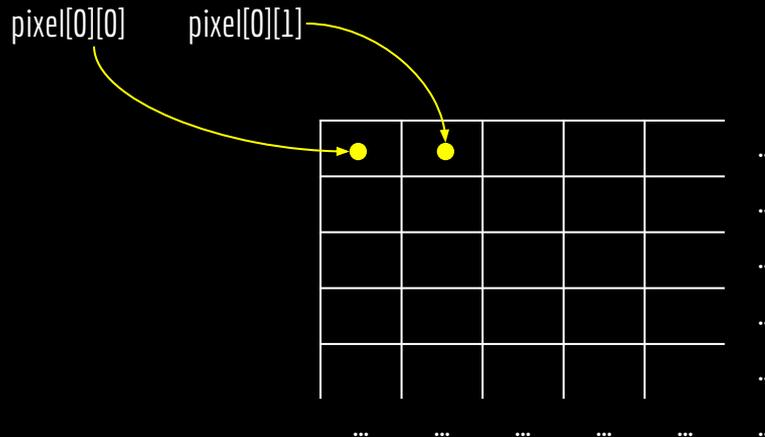
# Melhorando

É comum termos programas que executam a **mesma operação em múltiplos dados**.

Problemas clássicos envolvendo matrizes e vetores.

Exemplo: Deixar a imagem Full HD 10% mais escura (sem considerar as cores).

```
for(size_t i=0; i < 1080; ++i)
    for(size_t j=0; j < 1920; ++j)
        tela[i][j] = tela[i][j] * 0.9;
```



# SIMD

SIMD - Single instruction, Multiple Data.

Uma única instrução pode operar em múltiplos dados simultaneamente.

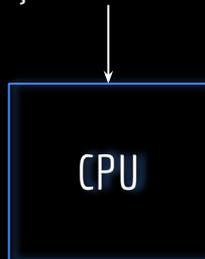
Exemplos: MMX e SSE da sua CPU x86-64.



Pentium 1 MMX (1997)

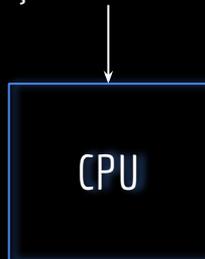
# SIMD

...  
Instrução4 Dado7 Dado8  
Instrução3 Dado5 Dado6  
Instrução2 Dado3 Dado4  
Instrução1 Dado1 Dado2

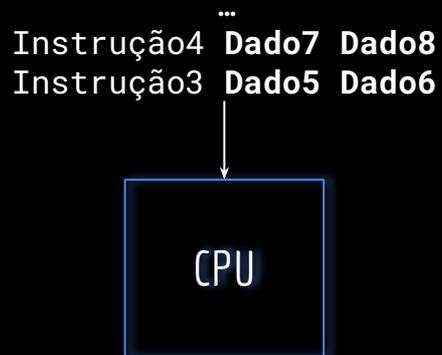


# SIMD

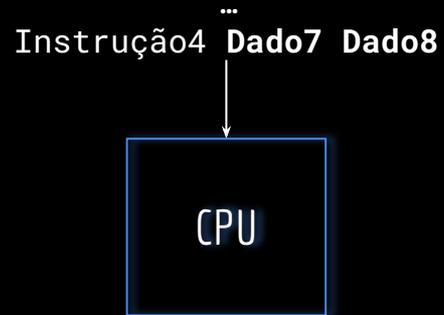
...  
Instrução4 Dado7 Dado8  
Instrução3 Dado5 Dado6  
Instrução2 Dado3 Dado4



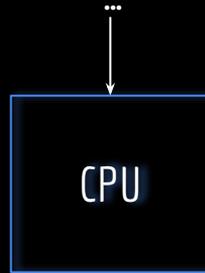
# SIMD



# SIMD



# SIMD



# Ainda Melhor!

Podemos colocar múltiplas CPUS (núcleos) SIMD em um único chip!

Temos múltiplas CPUs individuais.

- Múltiplas instruções ao mesmo tempo.

- Cada instrução opera em múltiplos dados.

MIMD - Multiple Instruction Multiple Data.

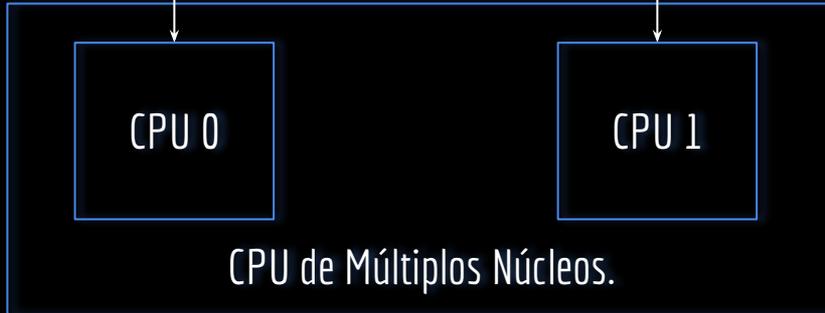
# MIMD

Processo/Thread 1

Processo/Thread 2

...  
Instrução4 **Dado7 Dado8**  
Instrução3 **Dado5 Dado6**  
Instrução2 **Dado3 Dado4**  
Instrução1 **Dado1 Dado2**

...  
Instrução4 **Dado7 Dado8**  
Instrução3 **Dado5 Dado6**  
Instrução2 **Dado3 Dado4**  
Instrução1 **Dado1 Dado2**



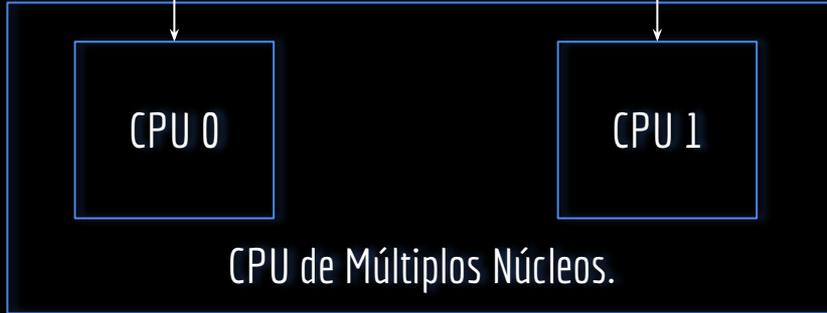
# MIMD

Processo/Thread 1

Processo/Thread 2

...  
Instrução4 **Dado7 Dado8**  
Instrução3 **Dado5 Dado6**  
Instrução2 **Dado3 Dado4**

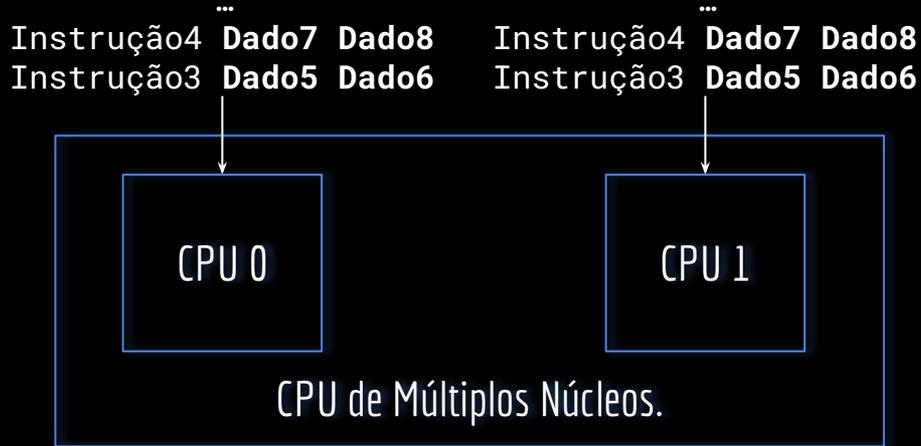
...  
Instrução4 **Dado7 Dado8**  
Instrução3 **Dado5 Dado6**  
Instrução2 **Dado3 Dado4**



# MIMD

Processo/Thread 1

Processo/Thread 2



# MIMD

Processo/Thread 1

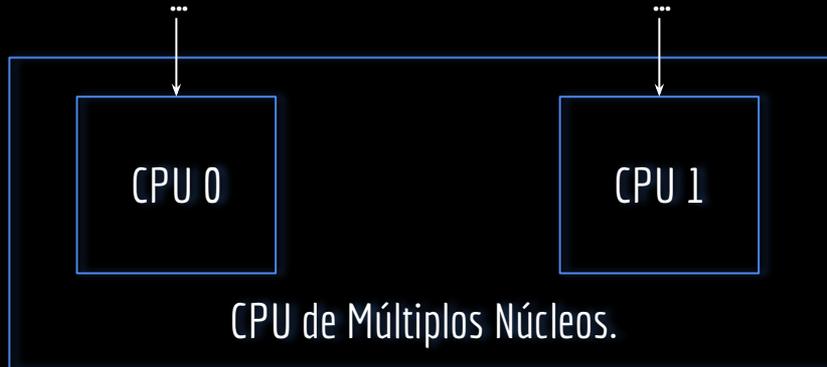
Processo/Thread 2



# MIMD

Processo/Thread 1

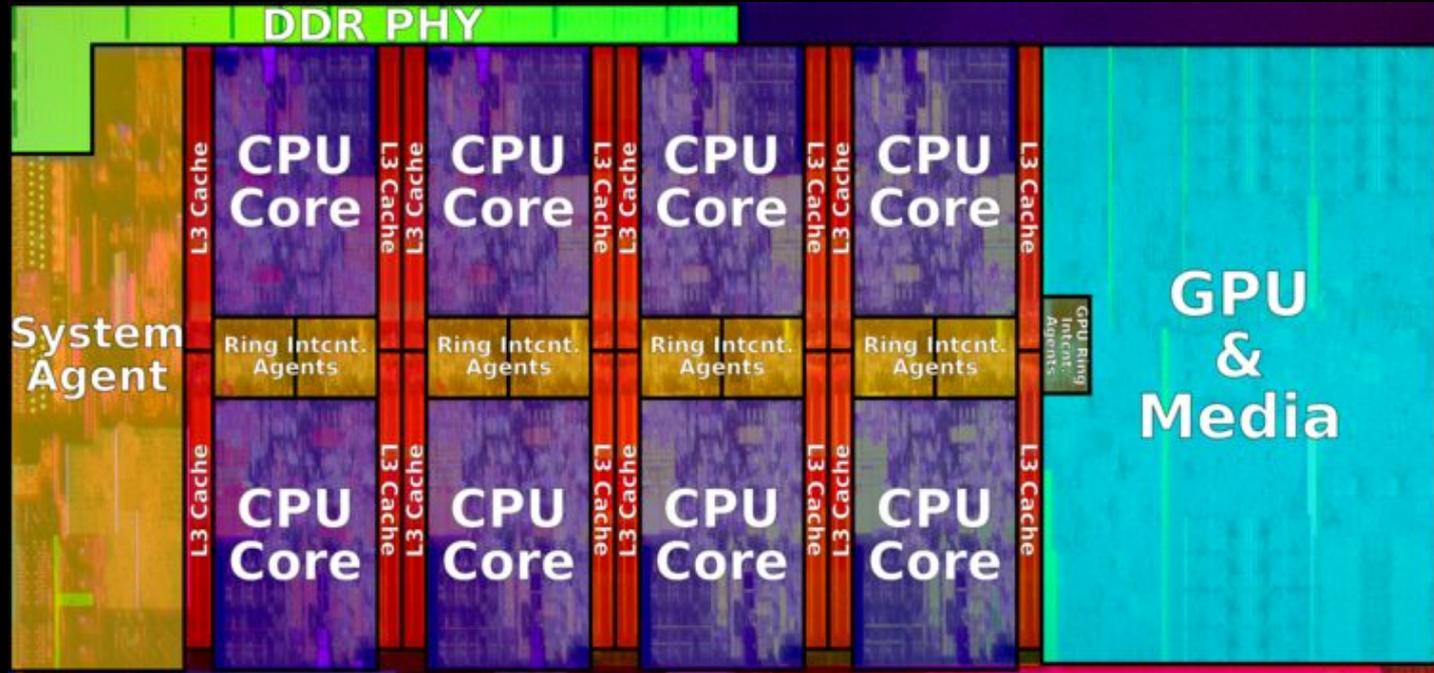
Processo/Thread 2



# MIMD



Intel i9-9900K

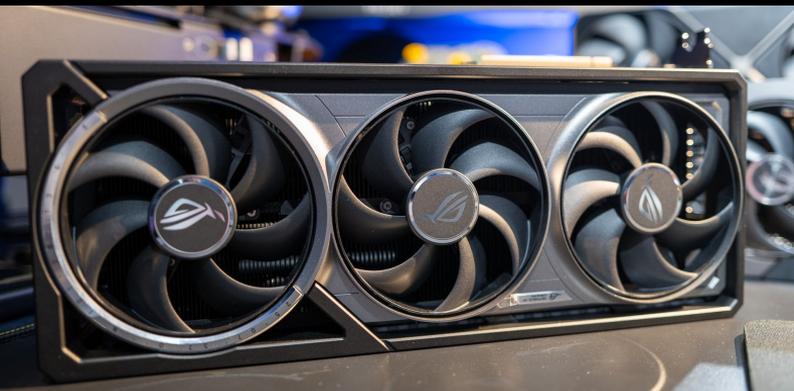
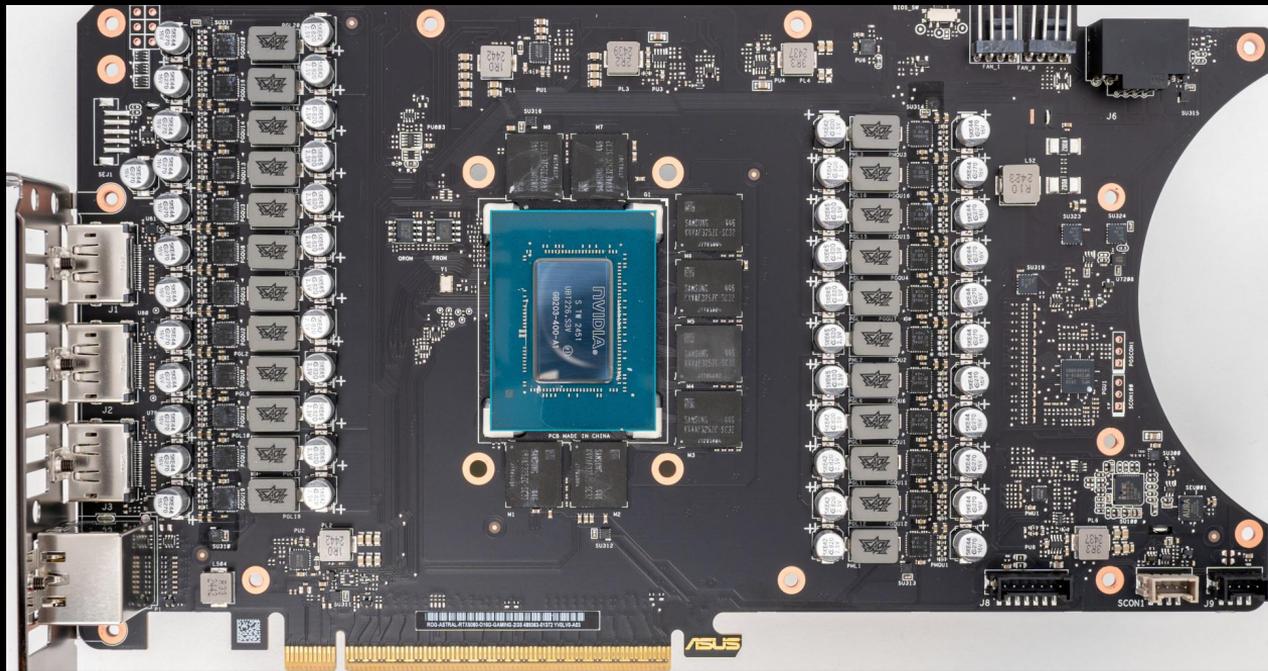


8 CPUs. Cada CPU é capaz de operar em paralelo em múltiplos dados.

# GPU

Graphics Processing Unit.

O que ela tem de tão especial quando comparada a uma CPU convencional?



Uma RTX 5080 sem a “casca”.

# RTX 5090



# GPU

Nasceram para realizar processamento gráfico.  
Indústria de jogos.

Hoje são largamente utilizadas também para “outras formas de computação”.

Exemplo: computação científica e simulações.



# GPU

Muitas operações em vídeo são uma sequência de operações em que o cálculo não muda.

Somente os operandos são diferentes.

Comumente operamos em matrizes.

A mesma operação é repetida para todos os elementos.

# GPU

Uma GPU é um tipo de CPU massivamente paralela.

Utiliza conceitos **SIMD** e **MIMD** largamente para realizar as tarefas repetitivas rapidamente.

# Compare



	Nvidia QUADRO GP100	Nvidia RTX 6000 Ada	Intel i7 14700
Lançamento	2016	2023	2024
Clock Base	1304 MHz	915 MHz	2100 MHz
Memória Principal	16 GB	48 GB	Até 192 GB
Barramento da Memória	717 GB/s	960 GB/s	89,6 GB/s
Consumo	235 W	300 W	65W
Núcleos (de acordo com o fabricante)	3584	18176	20

# GPUs

A GPU possui milhares de **processadores especializados**.

Capazes de executar muito rapidamente tarefas específicas, como somar os elementos de um vetor.

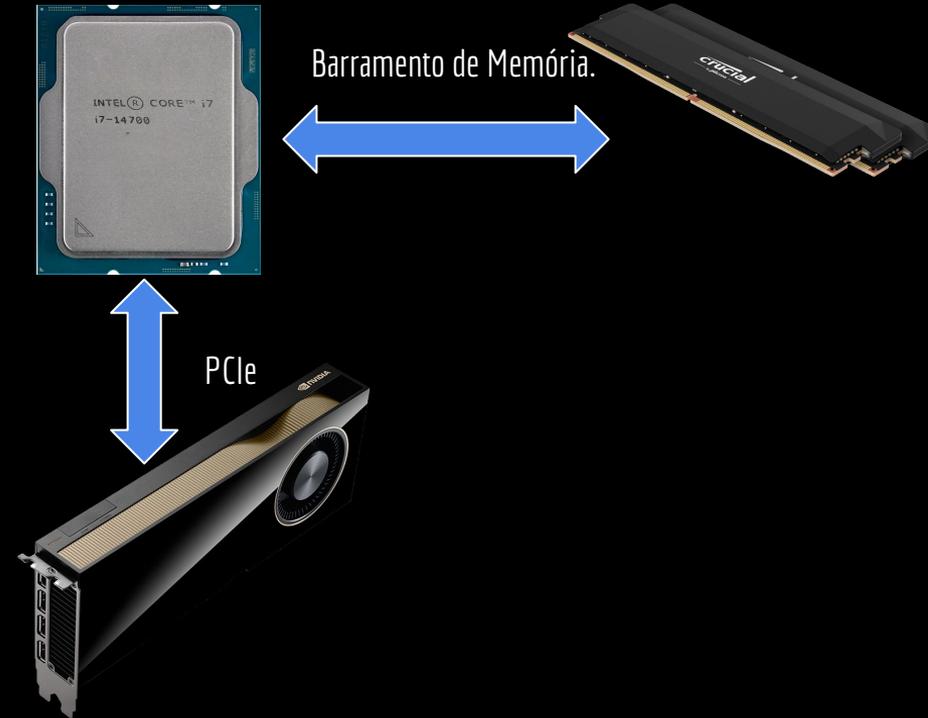
Os processadores da GPU **não são gerais o suficiente para executar qualquer tarefa de forma eficiente**.

# GPUs

Utilizamos a **CPU** como uma unidade de processamento geral e central.

A GPU opera de forma secundária.

Tarefas específicas e massivamente paralelizáveis.



# Arquitetura

Vamos utilizar uma **adaptação** das terminologias da Nvidia.

Tome as ideias de **maneira crítica**.

Muitas terminologias são confusas e dependentes do fabricante.

Considere ainda que essa é apenas uma introdução aos conceitos.

# Multithreaded SIMD processor

Uma GPU é composta de muitas CPUs com capacidades SIMD.

Cada CPU individual é chamada de **Multithreaded SIMD Processor**.

**MSP**

PCI Express 3.0 Host Interface

GigaThread Engine

High Bandwidth Memory 2

High Bandwidth Memory 2

High Bandwidth Memory 2

High Bandwidth Memory 2

L2 Cache

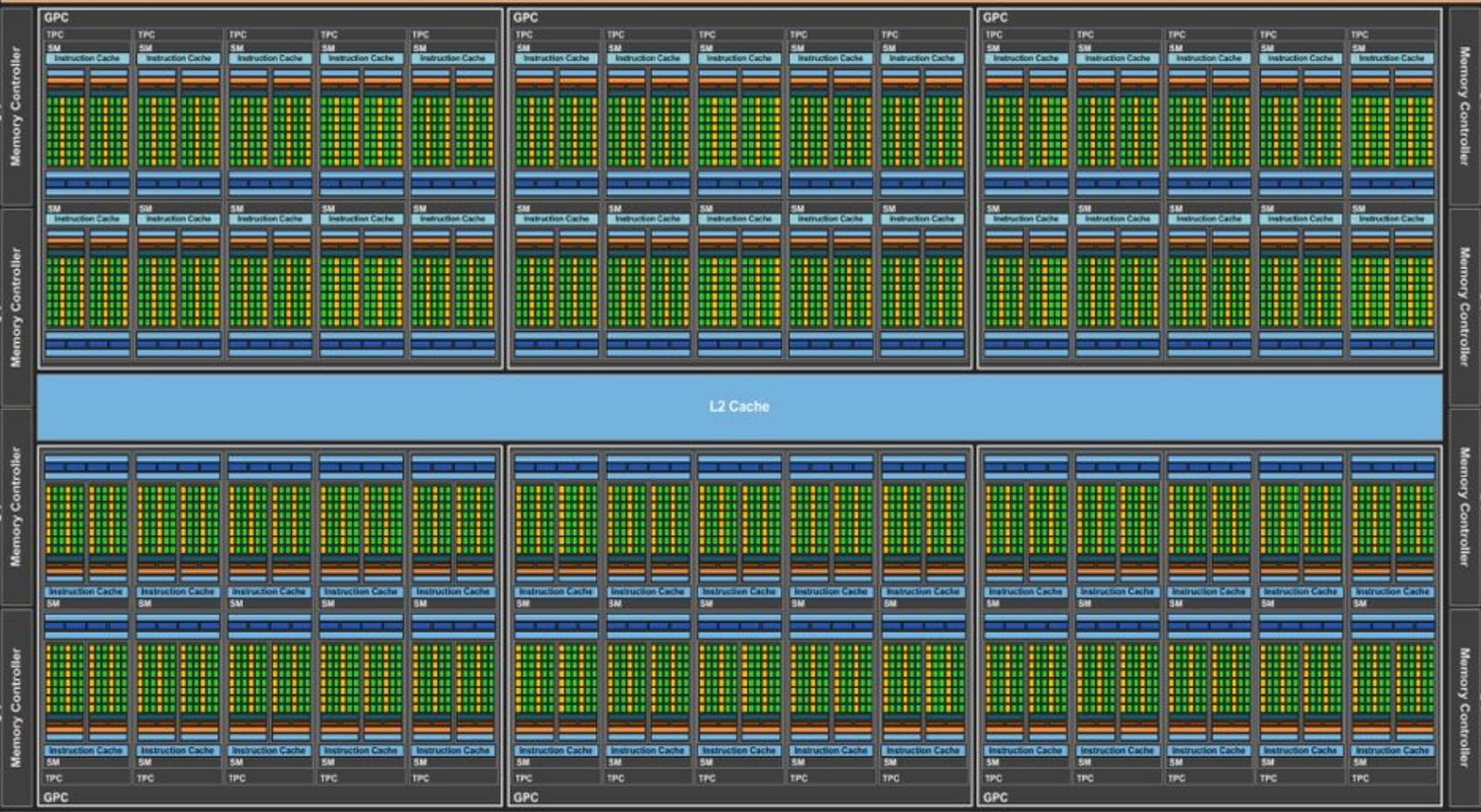
High-Speed Hub

NVLink

NVLink

NVLink

NVLink



# Arquitetura Nvidia Pascal

Obs.: No diagrama de blocos temos 60 MSPs. 56 arquitetura + 4 de reserva para falhas de fabricação.

A Pascal P100 possui 56.  
Multithreaded SIMD Processors

Temos então múltiplas CPUs, cada uma operando em dados diferentes.  
MIMD.



# Arquitetura Nvidia Pascal



# Multithreaded SIMD Processor - MSP



Cada Multithreaded SIMD Processor (MSP) pode despachar duas instruções SIMD.

“Multithreading”;

Cada instrução SIMD pode operar em até 32 elementos de uma vez.

# Multithreaded SIMD Processor - MSP



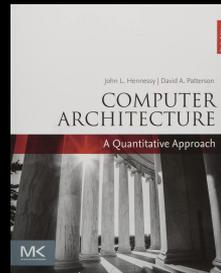
Cada Multithreaded SIMD Processor (MSP) pode despachar duas instruções SIMD.

“Multithreading”;

Cada instrução SIMD pode operar em até 32 elementos de uma vez.

O processo de execução das instruções é similar a um **processador vetorial**.

Leia sobre processadores vetoriais em: Hennessy, Patterson. Arquitetura de Computadores: uma abordagem quantitativa. 2019.



# Multithreaded SIMD Processor - MSP



Cada instrução pode operar em até 32 elementos de uma vez.

Mas temos apenas:

16 unidades para cálculo de ponto flutuante (DP).

8 Unidades para loads/stores (LD/ST).

8 Unidades para operações especiais (SFU).

Ex.: raiz quadrada.

Logo, pode ser necessário mais de um ciclo de clock para completar a operação (processamento vetorial).

# Multithreaded SIMD Processor - MSP



Sua GPU:

Possui múltiplos processadores (MSPs), cada um executando instruções diferentes em dados diferentes – **MIND**.

Cada MSP possui duas unidades de despacho de instruções – “Multithreading”.

Cada instrução pode executar a mesma operação em até 32 dados diferentes “ao mesmo tempo” – **SIMD**.

Possui paralelismo a nível de Instruções – **Pipeline**.

# GPU

Em uma GPU temos uma porção de tipos de paralelismo:

MIMD, SIMD, Multithreading e paralelismo a nível de instrução.

# Tome Cuidado

Analisando a GPU Pascal P100 do exemplo.

Temos 56 MSPs.





# Tome Cuidado

56 MSPs capazes de executar 2 instruções em paralelo cada.

$56 \times 2 = 112$  CPUS paralelas.

Especificação da Nvidia QUADRO GP100.

Largura de Banda da Memória

Até 717 GB/s

NVIDIA CUDA® Cores

3584

Interface do Sistema

PCI Express 3.0 x16

# Tome Cuidado

56 MSPs capazes de executar 2 instruções em paralelo cada.

$$56 \times 2 = 112 \text{ CPUS paralelas.}$$

Especificação da Nvidia QUADRO GP100.

Provavelmente por questões de marketing, a Nvidia multiplica por 32, já que cada instrução processa até 32 dados.

$$112 \times 32 = 3584$$

Em uma comparação mais justa, temos 112 “Cores SIMD”.

Vale lembrar ainda que uma CPU atual tenta despachar 4 instruções internamente.

Largura de Banda da Memória

Até 717 GB/s

NVIDIA CUDA® Cores

3584

Interface do Sistema

PCI Express 3.0 x16

# Escondendo Latência

Em sua “CPU Convencional”.

As instruções são executadas em ordem (ou quase isso).

**A memória principal é lenta.**

Para mitigar, temos caches grandes.

**O objetivo das caches é manter a CPU alimentada.**

# Escondendo Latência

Em uma GPU não faz sentido uma cache grande.

Uma cache grande o suficiente para comportar a carga de trabalho típica de uma GPU, escondendo a demora da memória principal, seria gigantesca.

Temos então uma **cache pequena** para armazenar alguns poucos dados de trabalho.



	Nvidia QUADRO GP100	Intel i7 14700
Cache	4 MB	33 MB

Obs.: contando apenas os níveis mais altos de cache.

# Escondendo Latência

Em uma GPU não faz sentido uma cache grande.

Uma cache grande o suficiente para comportar a carga de trabalho típica de uma GPU, escondendo a demora da memória principal, seria gigantesca.

Temos então uma **cache pequena** para armazenar alguns poucos dados de trabalho.



	Nvidia QUADRO GP100	Intel i7 14700
Cache	4 MB	33 MB
Largura de Banda de memória	717 GB/s	89,6 GB/s

Obs.: contando apenas os níveis mais altos de cache.

# Escondendo Latência

Em uma GPU não faz sentido uma cache grande.

Uma cache grande o suficiente para comportar a carga de trabalho típica de uma GPU, escondendo a demora da memória principal, seria gigantesca.

Temos então uma **cache pequena** para armazenar alguns poucos dados de trabalho.



	Nvidia QUADRO GP100	Intel i7 14700
Cache	4 MB	33 MB
Largura de Banda de memória	717 GB/s	89,6 GB/s

Obs.: contando apenas os níveis mais altos de cache.



# Escondendo Latência

Escondemos a latência (demora) da memória principal da GPU submetendo múltiplas instruções.

O escalonador sabe quais instruções já foram carregadas da memória.

Executa as instruções “prontas”, mesmo fora de ordem.

Ideia parecida com a explorada no algoritmo de Tomasulo.

# Escondendo Latência

Escondemos a latência (demora) da memória principal da GPU submetendo múltiplas instruções.

O escalonador sabe quais instruções já foram carregadas da memória.

Executa as instruções “prontas”, mesmo fora de ordem.

Ideia parecida com a explorada no algoritmo de Tomasulo.

**Não garantimos a ordem de execução.**

Somente que tudo será executado.

Os MSPs não ficam parados, desde que hajam instruções prontas.

# Exemplo



Escalonador SIMD

Escalonador SIMD

Unid. de Despacho

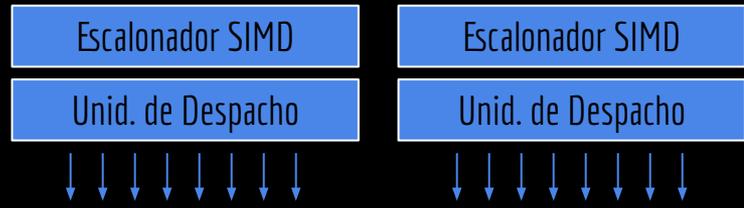
Unid. de Despacho

# Exemplo

Thread Block 0

```

...
não pronta   instrução SIMD 3 Thread 1
não pronta   instrução SIMD 2 Thread 1
não pronta   instrução SIMD 1 Thread 1
não pronta   instrução SIMD 0 Thread 1
não pronta   instrução SIMD 5 Thread 0
não pronta   instrução SIMD 4 Thread 0
não pronta   instrução SIMD 3 Thread 0
não pronta   instrução SIMD 2 Thread 0
não pronta   instrução SIMD 1 Thread 0
não pronta   instrução SIMD 0 Thread 0
    
```



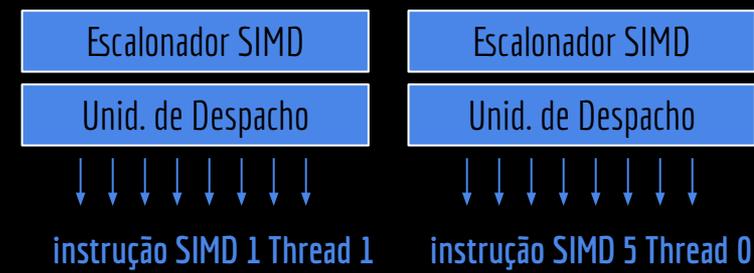
Não pronto  
**Pronto.**  
 Pronto e em execução.  
 Pronto e executado.

# Exemplo

Thread Block 0

```

...
não pronta   não pronta   não pronta   não pronta   não pronta   não pronta   não pronta
pronta       pronta       pronta       pronta       pronta       pronta       pronta
...
instrução SIMD 3 Thread 1
instrução SIMD 2 Thread 1
instrução SIMD 1 Thread 1
instrução SIMD 0 Thread 1
instrução SIMD 5 Thread 0
instrução SIMD 4 Thread 0
instrução SIMD 3 Thread 0
instrução SIMD 2 Thread 0
instrução SIMD 1 Thread 0
instrução SIMD 0 Thread 0
    
```



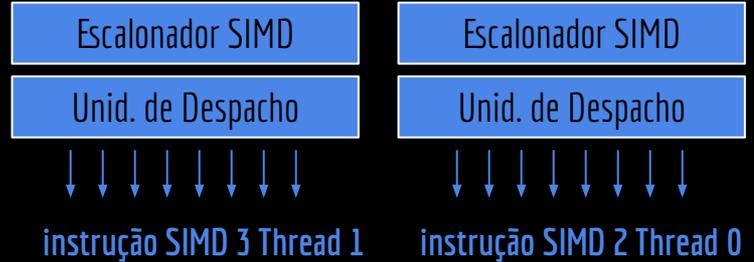
Não pronto  
**Pronto.**  
**Pronto e em execução.**  
**Pronto e executado.**

# Exemplo

Thread Block 0



...	pronta	instrução SIMD 3	Thread 1
...	não pronta	instrução SIMD 2	Thread 1
...	pronta	instrução SIMD 1	Thread 1
...	não pronta	instrução SIMD 0	Thread 1
...	pronta	instrução SIMD 5	Thread 0
...	não pronta	instrução SIMD 4	Thread 0
...	não pronta	instrução SIMD 3	Thread 0
...	pronta	instrução SIMD 2	Thread 0
...	pronta	instrução SIMD 1	Thread 0
...	pronta	instrução SIMD 0	Thread 0



Não pronto

Pronto.

Pronto e em execução.

Pronto e executado.

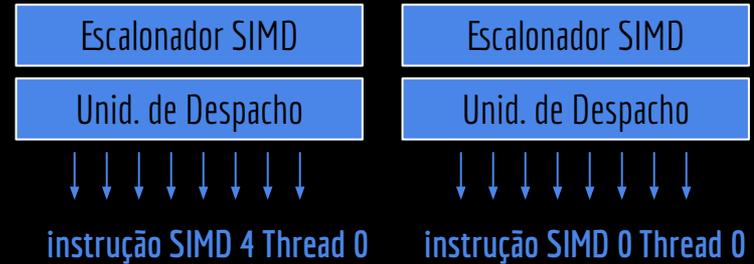
# Exemplo

Thread Block 0



```

...
pronta          instrução SIMD 3 Thread 1
pronta          instrução SIMD 2 Thread 1
pronta          instrução SIMD 1 Thread 1
não pronta     instrução SIMD 0 Thread 1
pronta          instrução SIMD 5 Thread 0
pronta          instrução SIMD 4 Thread 0
pronta          instrução SIMD 3 Thread 0
pronta          instrução SIMD 2 Thread 0
pronta          instrução SIMD 1 Thread 0
pronta          instrução SIMD 0 Thread 0
    
```



Não pronto

**Pronto.**

**Pronto e em execução.**

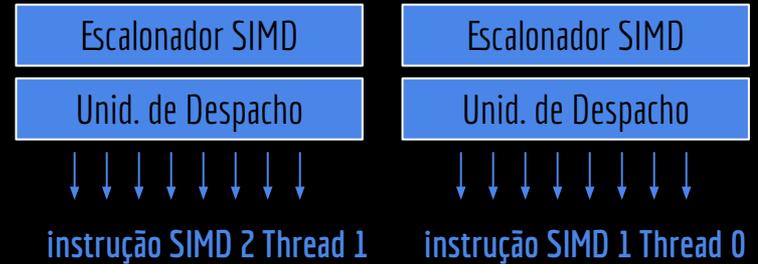
**Pronto e executado.**

# Exemplo

Thread Block 0

```

...
pronta          instrução SIMD 3 Thread 1
pronta          instrução SIMD 2 Thread 1
pronta          instrução SIMD 1 Thread 1
não pronta     instrução SIMD 0 Thread 1
pronta          instrução SIMD 5 Thread 0
pronta          instrução SIMD 4 Thread 0
pronta          instrução SIMD 3 Thread 0
pronta          instrução SIMD 2 Thread 0
pronta          instrução SIMD 1 Thread 0
pronta          instrução SIMD 0 Thread 0
    
```



Não pronto

Pronto.

Pronto e em execução.

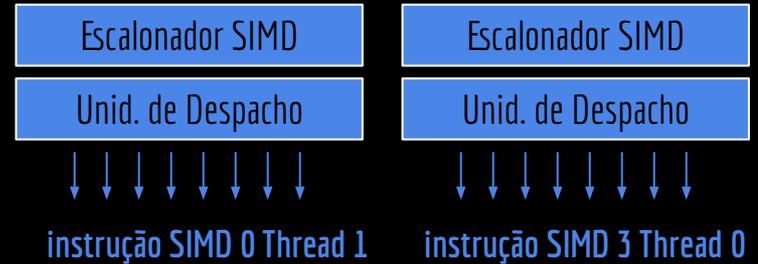
Pronto e executado.

# Exemplo

Thread Block 0

```

...
pronta          instrução SIMD 3 Thread 1
pronta          instrução SIMD 2 Thread 1
pronta          instrução SIMD 1 Thread 1
pronta          instrução SIMD 0 Thread 1
pronta          instrução SIMD 5 Thread 0
pronta          instrução SIMD 4 Thread 0
pronta          instrução SIMD 3 Thread 0
pronta          instrução SIMD 2 Thread 0
pronta          instrução SIMD 1 Thread 0
pronta          instrução SIMD 0 Thread 0
    
```



Não pronto

**Pronto.**

**Pronto e em execução.**

**Pronto e executado.**



# Divergência de WARP

Uma GPU **não** sabe fazer ifs.

Em cada fluxo de instruções SIMD – chamado de **WARP**, as instruções devem ser executadas em sequência.

O MSP que executa o WARP está em **lock step**.

Executa a mesma operação para todos os 32 dados.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture>

# Divergência de WARP

Uma GPU **não** sabe fazer ifs.

Em cada fluxo de instruções SIMD – chamado de **WARP**, as instruções devem ser executadas em sequência.

O MSP que executa o WARP está em **lock step**.

Executa a mesma operação para todos os 32 dados.

Problema:

```
...  
if(i%2 != 0)  
    a[i] = b[i]*c[i];  
else  
    a[i] = b[i]/c[i];  
...
```



# Divergência de WARP

Isso é uma **divergência de WARP**.

Em média, ao se inserir um `if/else`, o desempenho cai em 50%.

Se aninhar um `if`, o desempenho cai para 25%.

Se aninhar mais uma vez, cai para 12,5%.

...

# Divergência de WARP

Em Resumo:

GPU não sabe fazer ifs e elses.

Evite desvios.

É comum desvios serem gerados por **condições de borda**.

Exemplo: o cálculo das extremidades de uma matriz pode ser diferente do cálculo feito para os demais itens.

Solução: Fazer o cálculo para os elementos centrais, e depois fazer o cálculo para os elementos das bordas.

# Criando programas

Geralmente contamos com extensões das linguagens C/C++.

Para o desenvolvimento de Jogos:

- Pixel Shaders (Calcular a luz nos elementos).

  - High-Level Shading Language - HLSL.

  - C for Graphics - Cg.

- APIs de alto nível.

  - OpenGL (Open Source).

  - Direct3D (Microsoft e Parceiros).

Para computação científica.

- OpenCL (Open Source).

- CUDA (Nvidia).

# Exemplo

Considere o programa em C para somar o conteúdo de dois vetores, e armazenar o resultado no primeiro.

```
void soma(double* v1, double* v2, int tamanho){  
    for(int i=0; i < tamanho; i++)  
        v1[i] = v1[i] + v2[i];  
}
```

# Exemplo - CUDA

```
__global__  
void soma(double* v1, double* v2, int tamanho){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    v1[i] = v1[i] + v2[i];  
}
```

```
__host__  
int nblocks = (tamanho+255)/256;  
//Para chamar a função  
soma<<<nblocks,256>>>();
```

# Exemplo - CUDA

Global é a memória de vídeo.

```
__global__  
void soma(double* v1, double* v2, int tamanho){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    v1[i] = v1[i] + v2[i];  
}
```

```
__host__  
int nblocks = (tamanho+255)/256;  
//Para chamar a função  
soma<<<nblocks, 256>>>();
```

Host indica a memória principal da máquina.

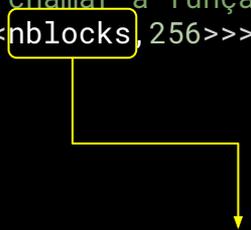
Considerando que cada bloco de threads vai operar em 256 elementos.

# Exemplo - CUDA

Exemplo: considere tamanho = 512.

```
__global__  
void soma(double* v1, double* v2, int tamanho){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    v1[i] = v1[i] + v2[i];  
}
```

```
__host__  
int nblocks = (tamanho+255)/256;  
//Para chamar a função  
soma<<<nblocks, 256>>>();
```



$nblocks = (512 + 255) / 256 = 2$  blocos.

# Exemplo - CUDA

Exemplo: considere tamanho = 512.

```
__global__  
void soma(double* v1, double* v2, int tamanho){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    v1[i] = v1[i] + v2[i];  
}
```

```
__host__  
int nblocks = (tamanho+255)/256;  
//Para chamar a função  
soma<<<nblocks,256>>>();
```

Grid

Warp 0  $v1[0] = v1[0]+v2[0]$   
...  
 $v1[31] = v1[31]+v2[31]$

Bloco 0 :

Warp 8  $v1[224] = v1[224]+v2[224]$   
...  
 $v1[255] = v1[255]+v2[255]$

Warp 0  $v1[256] = v1[256]+v2[256]$   
...  
 $v1[287] = v1[287]+v2[287]$

Bloco 1 :

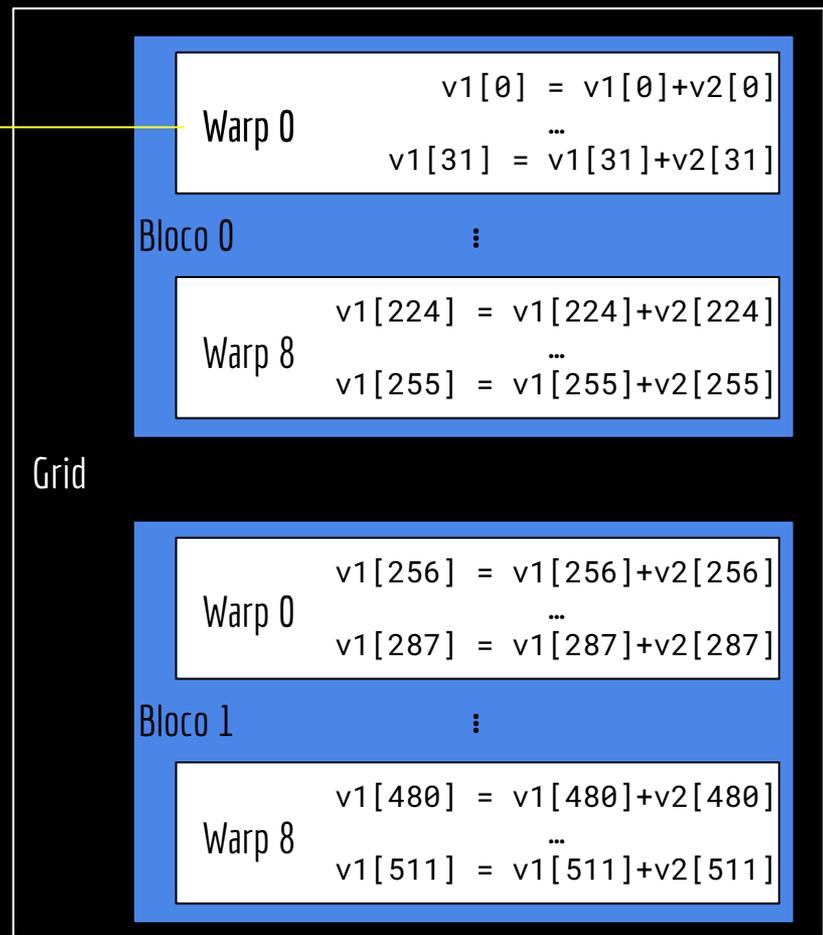
Warp 8  $v1[480] = v1[480]+v2[480]$   
...  
 $v1[511] = v1[511]+v2[511]$

# Exemplo - CUDA

Exemplo: considere tamanho = 512.

```
__global__  
void soma(double* v1, double* v2, int tamanho){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    v1[i] = v1[i] + v2[i];  
}
```

```
__host__  
int nblocks = (tamanho+255)/256;  
//Para chamar a função  
soma<<<nblocks,256>>();
```



Instrução SIMD Warp0:  $v1[0] = v1[0]+v2[0]$  |  $v1[1] = v1[1]+v2[1]$  | ... |  $v1[31] = v1[31]+v2[31]$

# Concluindo

Boa parte dos slides são aproximações (muitas vezes grosseiras) do que realmente acontece em uma GPU.

- Facilitar o entendimento da arquitetura.

- Tome os conceitos com cautela.

Boa parte dos slides se baseia na arquitetura Pascal P100.

- GPUs diferentes podem ter diferentes:

  - Tamanhos de instrução SIMD.

  - Número de MSPs.

  - Número de unidades funcionais em cada MSP.

  - ...

  - Você precisa considerar tudo isso ao criar seus programas!

# Concluindo

**Criar um programa para uma GPU não é fácil!**

Executar bibliotecas/funcões prontas que implementam programas dentro das nossas GPUs é fácil.

Encontrar alguém que realmente saiba criar um programa para GPUs é difícil.

Primeiro entenda a arquitetura.

Difere de muitas formas de um programa para CPU.

Se torna mais claro quando você entende as peculiaridades das GPUs.

# Exercícios

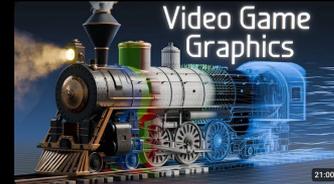
1. Pesquise no site da Nvidia as configurações de uma placa de vídeo de sua preferência. Verifique.
  - a. O modelo do processador.
  - b. Quantos Processadores a placa possui.
  - c. Quantos cores essa placa realmente possui. Você pode assumir que as instruções SIMD operam em 32 dados como na arquitetura PASCAL. Ou melhor, pesquise quantos dados realmente são processados por cada instrução SIMD na placa que você pesquisou.
2. Instale a API CUDA ou OpenCL em sua máquina, e crie um programa simples para somar ou multiplicar vetores, como visto em aula.
3. Nos últimos anos a NVidia e afins têm utilizado outras nomenclaturas para suas unidades de processamento, como os Tensor Cores. O que é um Tensor Core? Como é a arquitetura? O que ele faz de diferente? Aliás, o que é um Tensor?

# Exercícios

4. Assista ao vídeo: <https://youtu.be/h9Z4oGN89MU?si=YEn9xn1IFAh44W6>



5. Assista ao vídeo: <https://youtu.be/C8YtdC8mxTU?si=KafIISNAshAqLNLp>



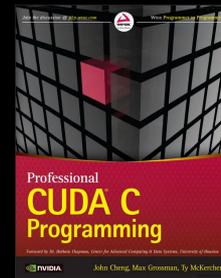
6. Assista ao vídeo: <https://youtu.be/iOlehM5kNSk?si=Wrl0GLj6gKOR8ZDr>



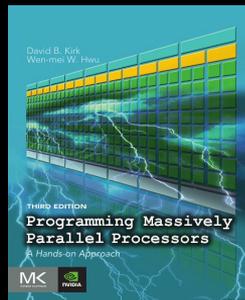
# Referências



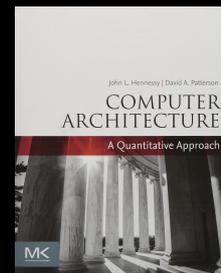
Cheng; Grossman; McKercher.  
Professional CUDA C  
Programming. 2014.



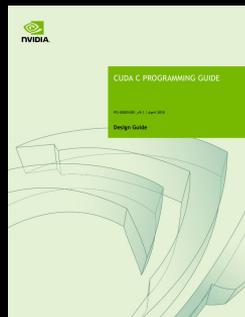
Kirk, Hwu. Programming  
Massively Parallel Processors:  
A Hands-on Approach. 3a  
edição. 2016.



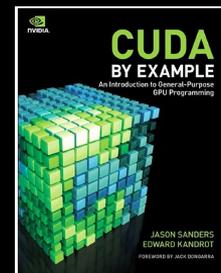
Hennessy, Patterson.  
Arquitetura de Computadores:  
uma abordagem quantitativa.  
2019.



Nvidia. CUDA C  
Programming Guide. 2018.



Sanders; Kandrot. CUDA by  
Example. 2010.



# Licença

Esta obra está licenciada com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).