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

Paulo Ricardo Lisboa de Almeida

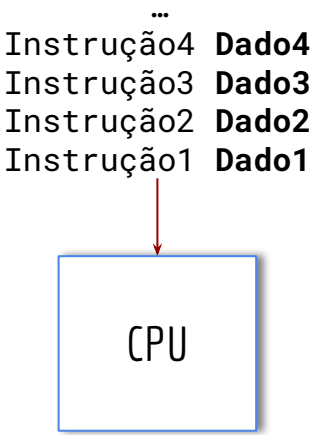
CPU

Vamos aos conceitos básicos.

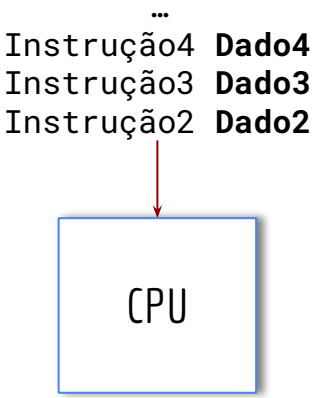
CPU de apenas “um núcleo”, que processa as instruções uma a uma.

- Cada instrução opera em um dado.
- SISD - Single Instruction, Single Data.

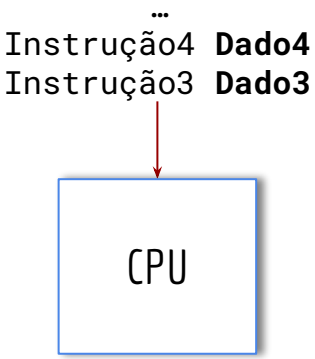
CPU


...
Instrução4 Dado4
Instrução3 Dado3
Instrução2 Dado2
Instrução1 Dado1
CPU

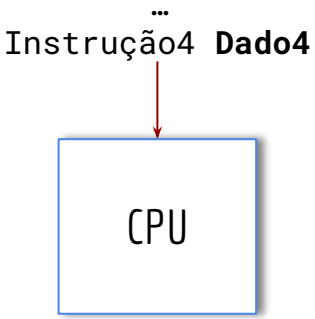
CPU



CPU



CPU



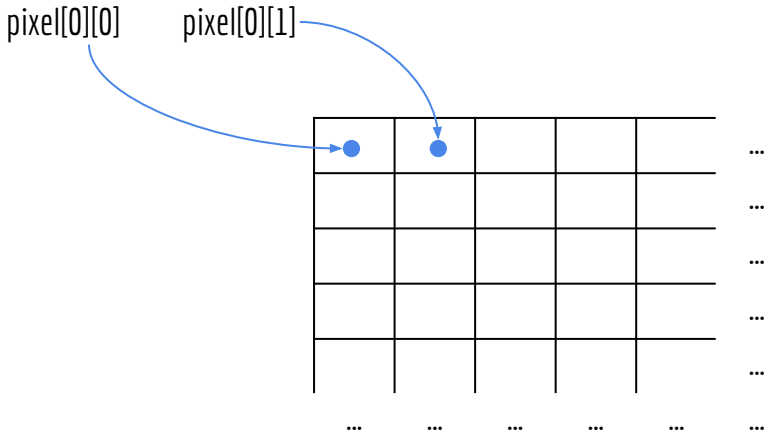
Podemos melhorar

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



SIMD

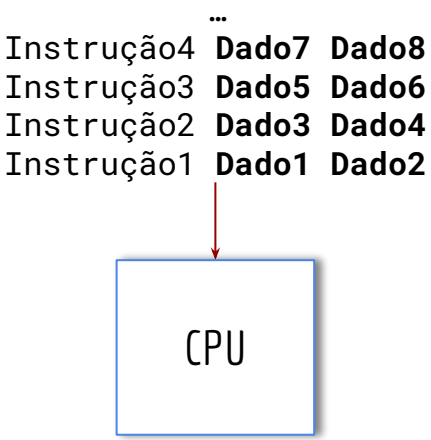
SIMD – Single instruction, Multiple Data.

Uma única instrução pode operar em múltiplos dados simultaneamente.

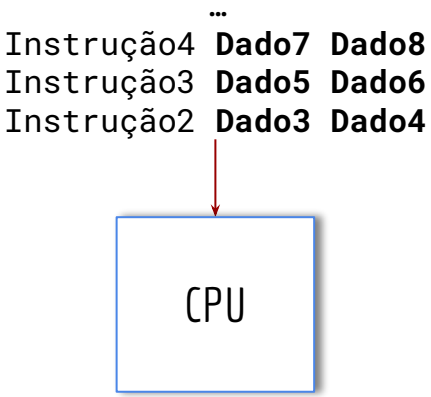
Exemplos: MMX e SSE da sua CPU x86-64.

Pentium 1 MMX (1997)

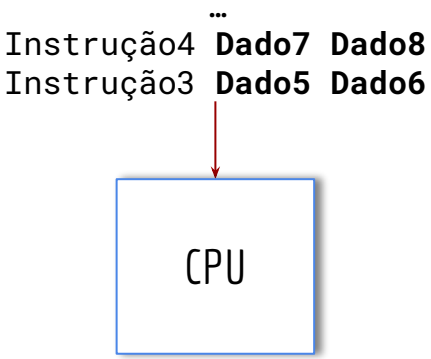
CPU


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

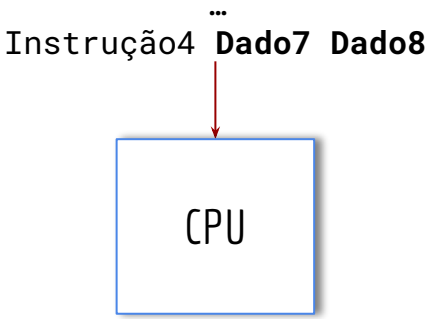
CPU


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

CPU



CPU



Podemos fazer ainda mais!

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.

CPU

Processo/Thread 1

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

Processo/Thread 2

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

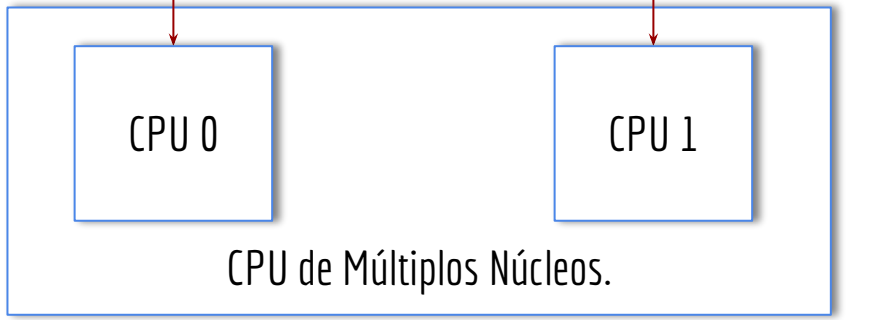
CPU

Processo/Thread 1

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

Processo/Thread 2

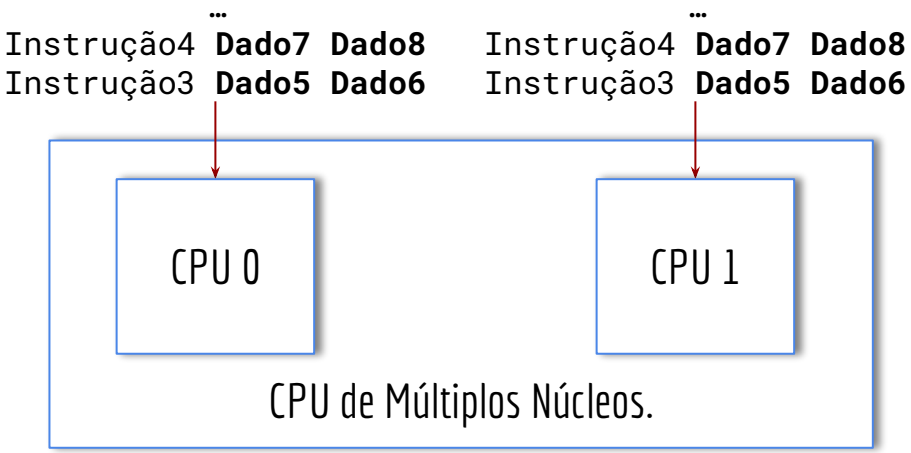
```
...
Instrução4 Dado7 Dado8
Instrução3 Dado5 Dado6
Instrução2 Dado3 Dado4
```



CPU

Processo/Thread 1

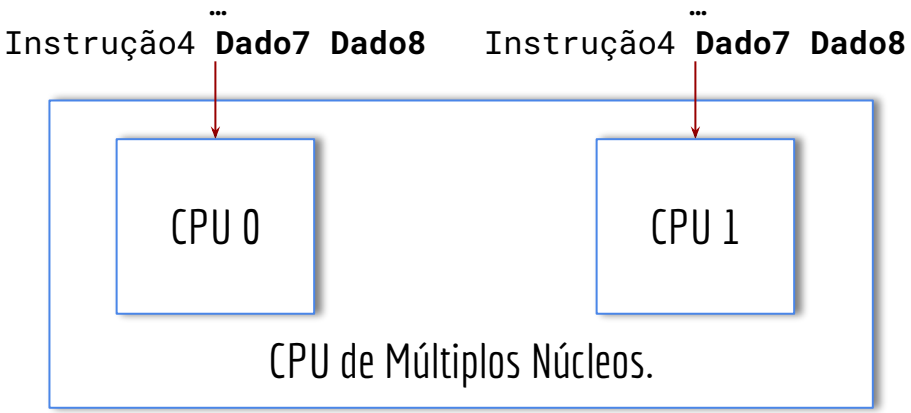
Processo/Thread 2



CPU

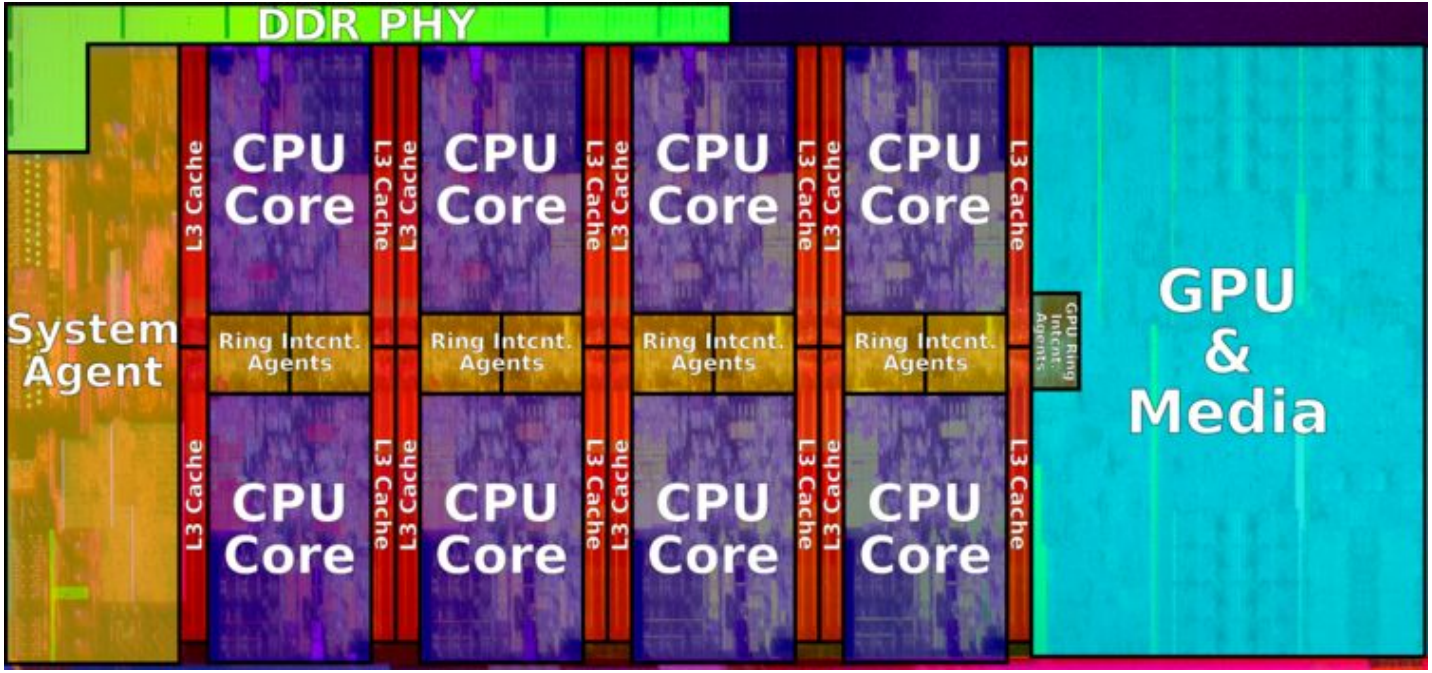
Processo/Thread 1

Processo/Thread 2



MIMD

Intel i9-9900K



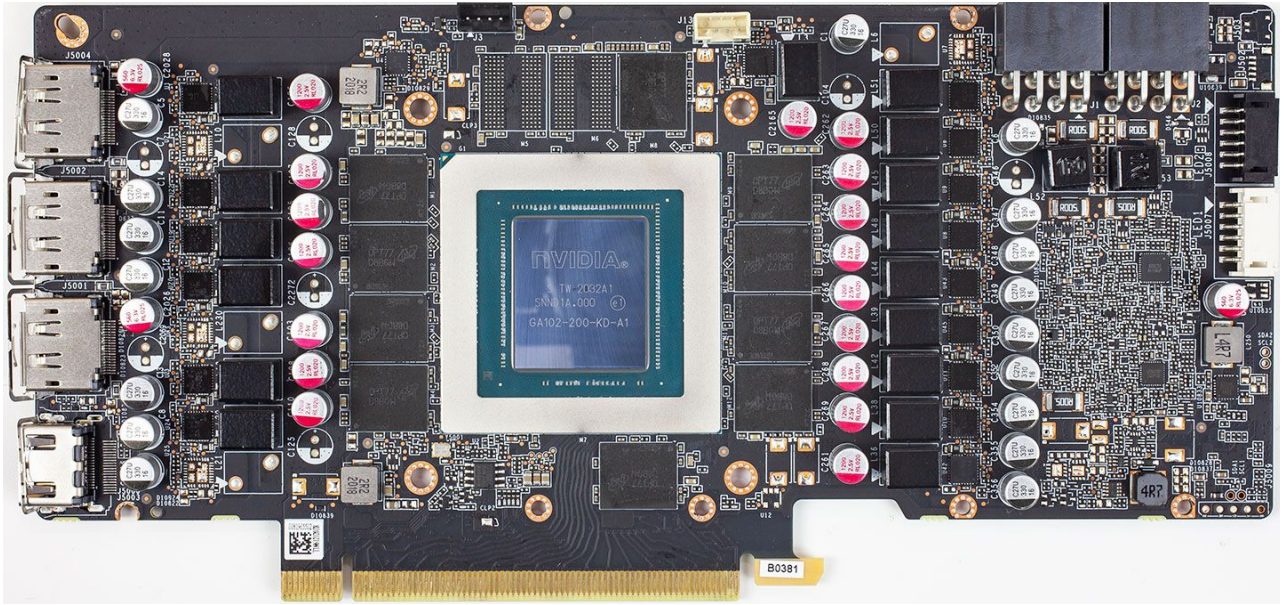
8 CPUs. Cada CPU é capaz de operar em paralelo em múltiplos dados.

GPU

GPU - Graphics Processing Unit.

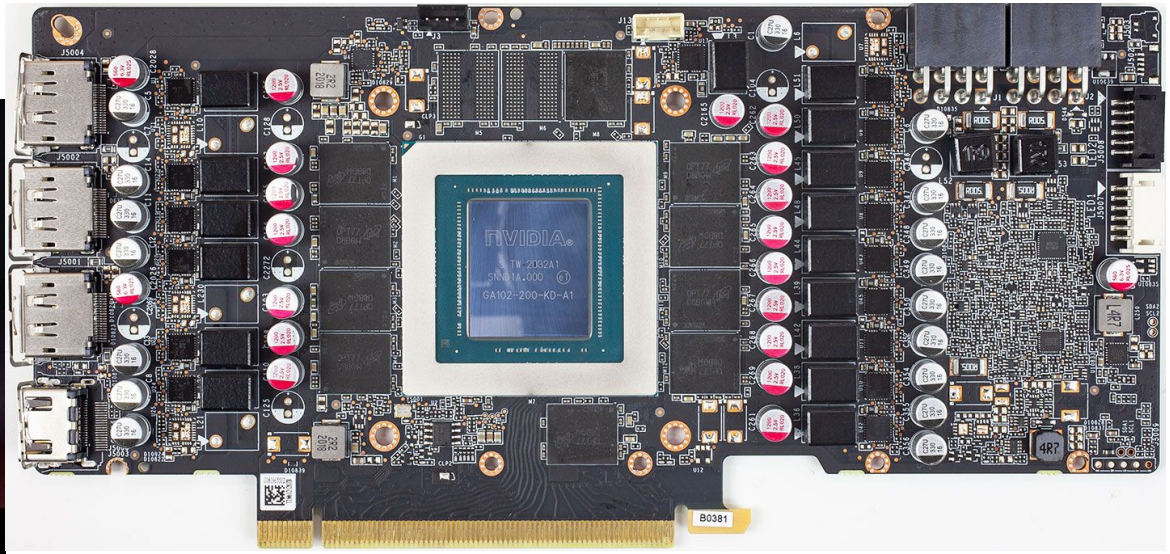
Unidade de Processamento Gráfico.

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



Uma RTX 3080 sem a “casca”.


ROG STRIX
GEFORCE RTX


NVIDIA
GA102-200-KD-A1
B0381


LOOK AT ME, I AM THE MOTHERBOARD NOW
MemeZila.com

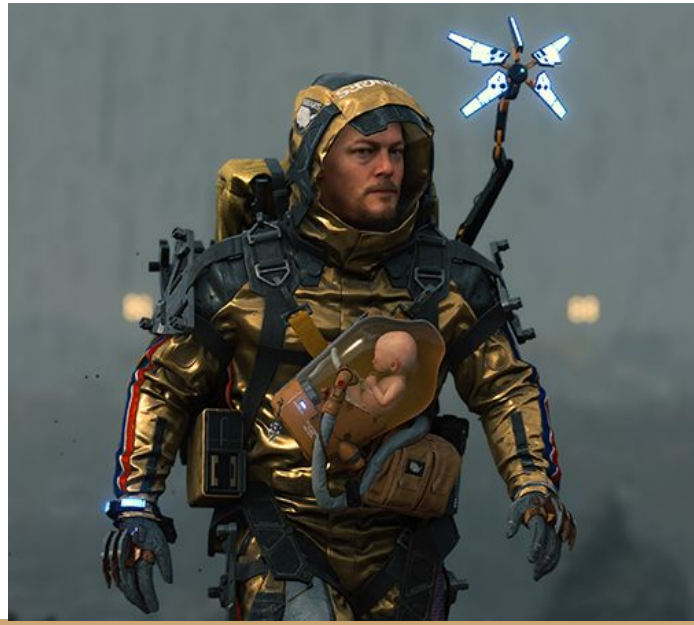
GPU

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” relacionadas à vídeo mais rapidamente.

Compare

	Nvidia QUADRO GP100	Intel i9 9900k
Clock Base	1304 MHz	3600MHz
Memória Principal	16 GiB	Até 128 GiB
Barramento da Memória	717 GiB/s	41,6 GiB/s
Consumo	235W	95W
Núcleos (de acordo com o fabricante)	3584	8

Compare

Se a GPU tem tantos núcleos a mais, deve haver algum motivo para ainda termos uma “CPU convencional”.

	Nvidia QUADRO GP100	Intel i9 9900k
Clock Base	1304 MHz	3600MHz
Memória Principal	16 GiB	Até 128 GiB
Barramento da Memória	717 GiB/s	41,6 GiB/s
Consumo	235W	95W
Núcleos (de acordo com o fabricante)	**3584**	**8**

GPUs

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

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

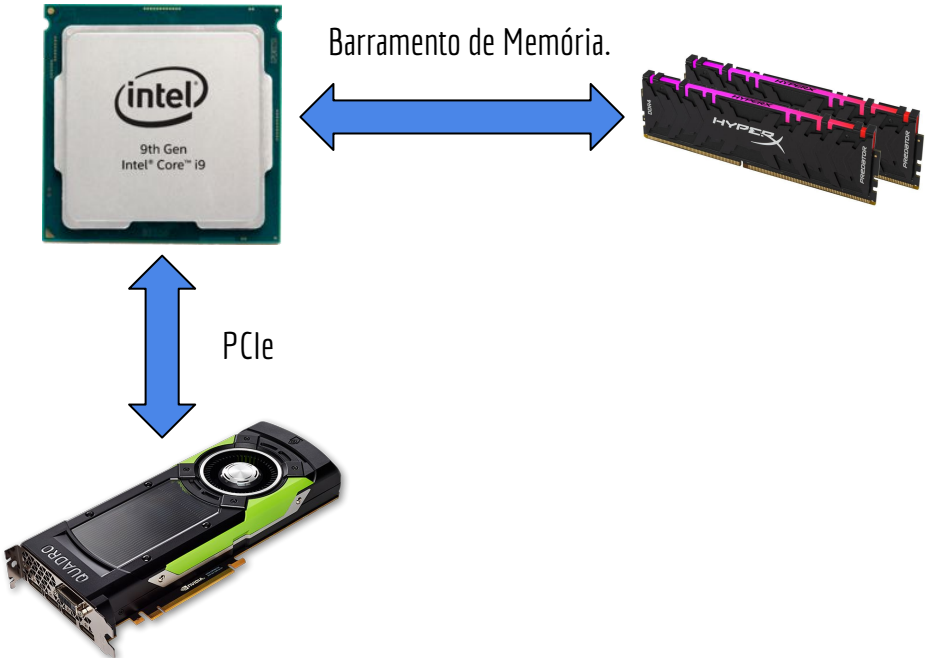
Os processadores da GPU **não são gerais o suficiente para executar qualquer tarefa de forma eficiente.**

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** terminologias da Nvidia.

Tome as ideias de **maneira crítica.**

Muitas terminologias são confusas e dependentes do fabricante.

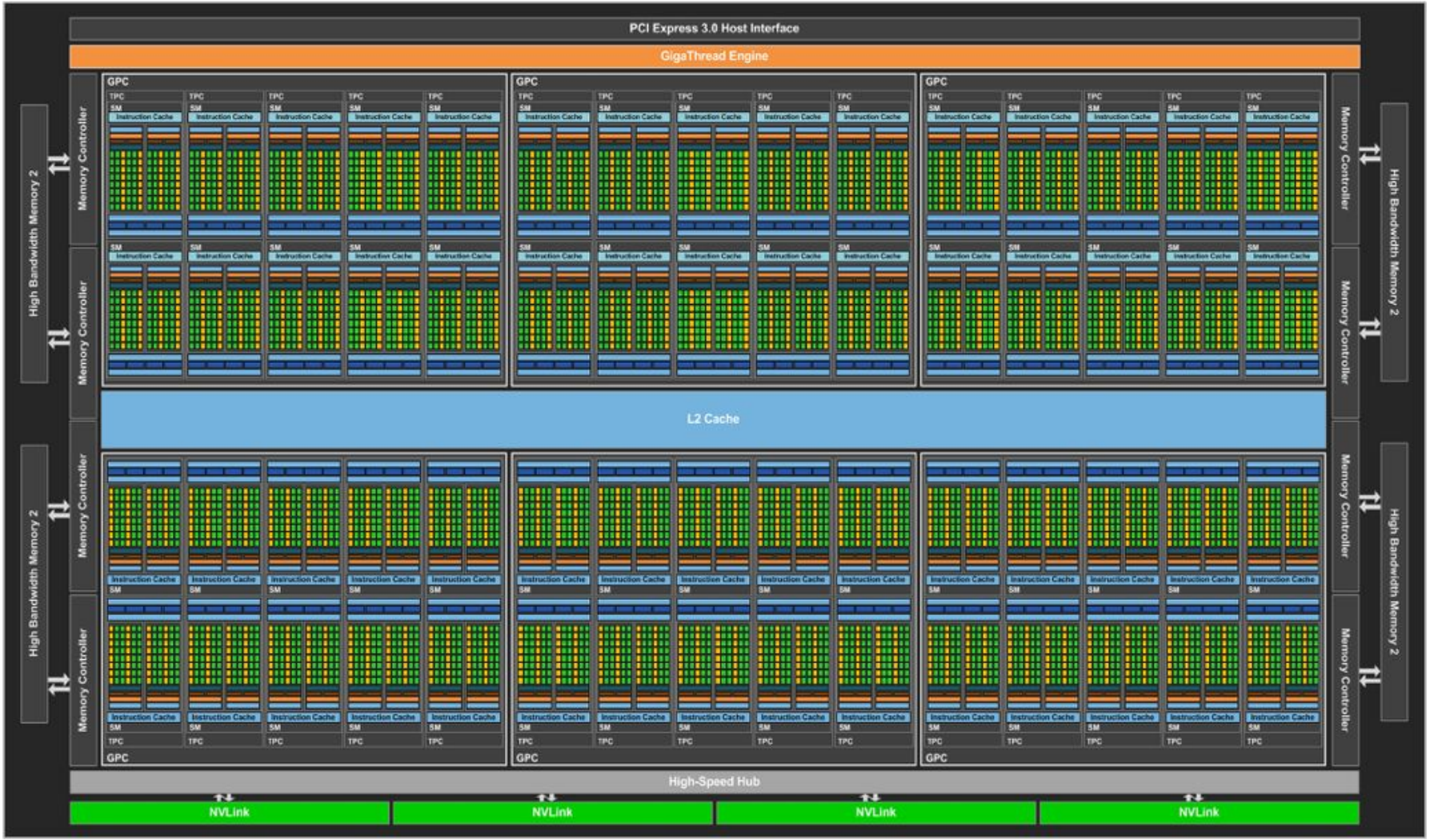
Considere ainda que essa é apenas uma introdução aos conceitos.

Multithreaded SIMD processor

Uma GPU é composta de muitas CPUs com capacidades SIMD.

Cada CPU individual é chamada de **M**ultithreaded **S**IMD **P**rocessor.

MSP


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink
NVLink
NVLink
NVLink

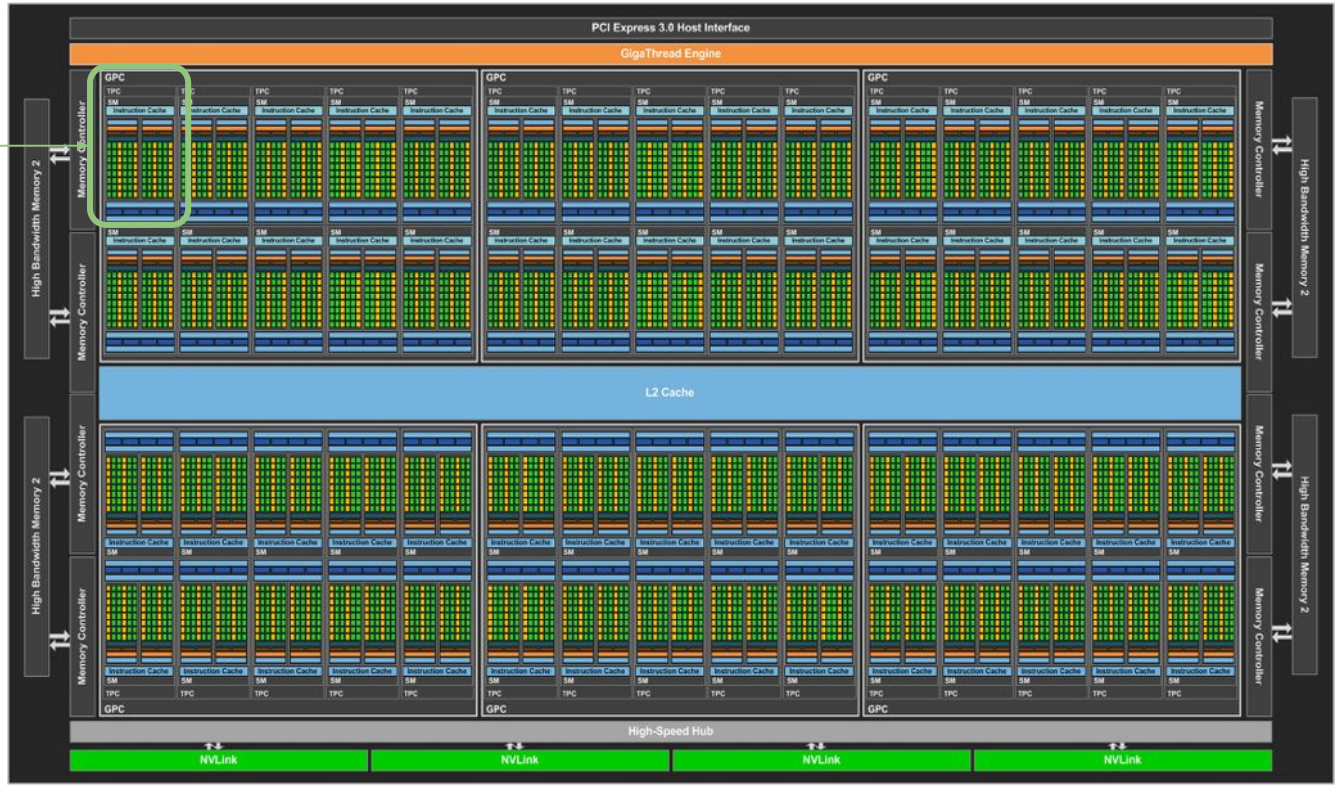
Arquitetura Nvidia Pascal

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

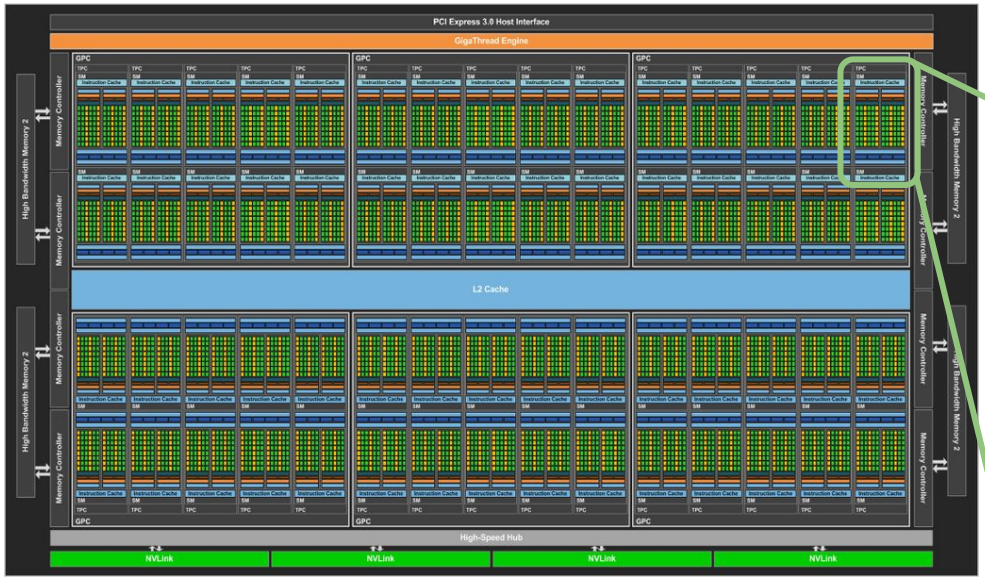
A Pascal P100 possui 56 Multithreaded SIMD Processors.

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:

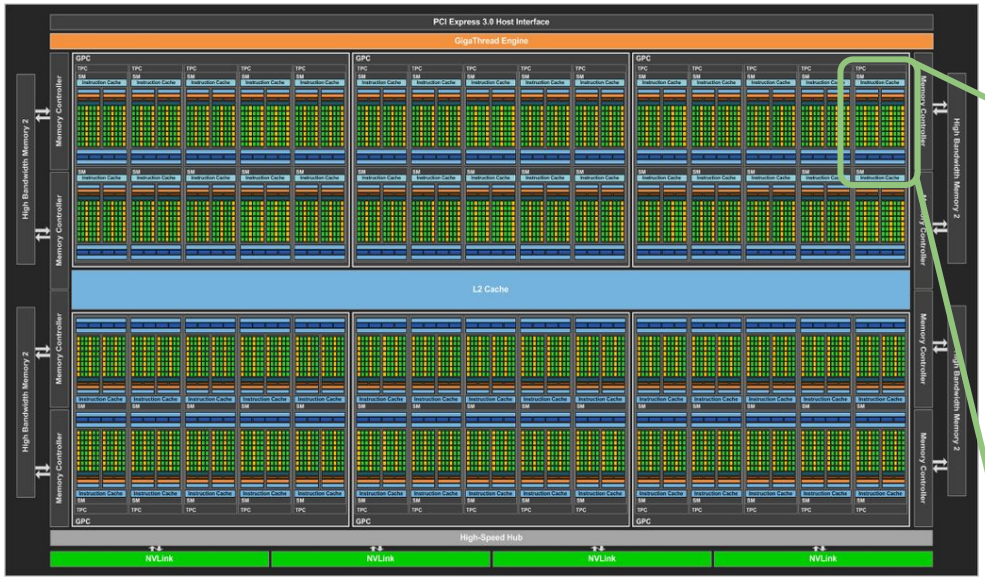
MIMD, SIMD, Multithreading e paralelismo a nível de instrução.

Tome Cuidado

Analisando a GPU Pascal P100 do exemplo.

Temos 56 MSPs.

Tome Cuidado



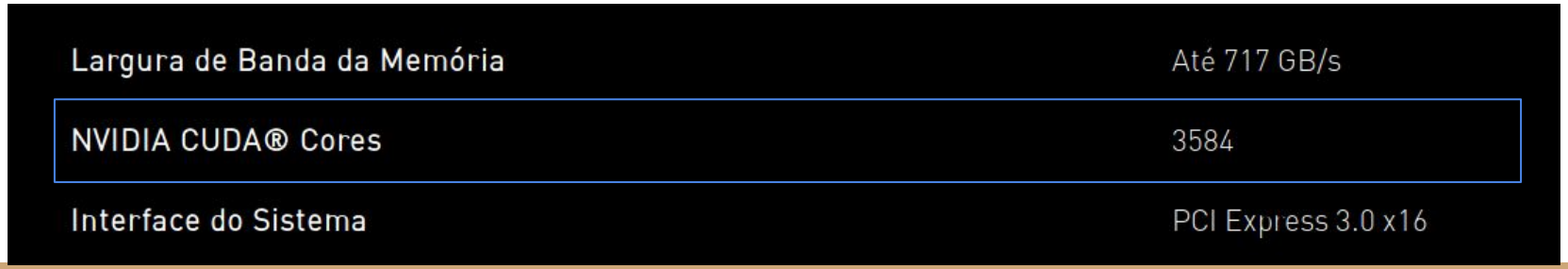
Cada MSP é capaz de despachar duas instruções SIMD ao mesmo tempo.
"Duas CPUs" em cada MSP: 56x2 = **112 CPUS paralelas.**

Tome Cuidado

56 MSPs capazes de executar 2 instruções em paralelo cada.

56 x 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 x 2 = **112 CPUS paralelas.**

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

Especificação da Nvidia QUADRO GP100.

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

112 x 32 = 3584

Em uma comparação mais justa, temos 112 "Cores SIMD".

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 Normal”.

- 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 i9 9900k
Cache	4 MiB	16 MiB

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

Escondendo Latência

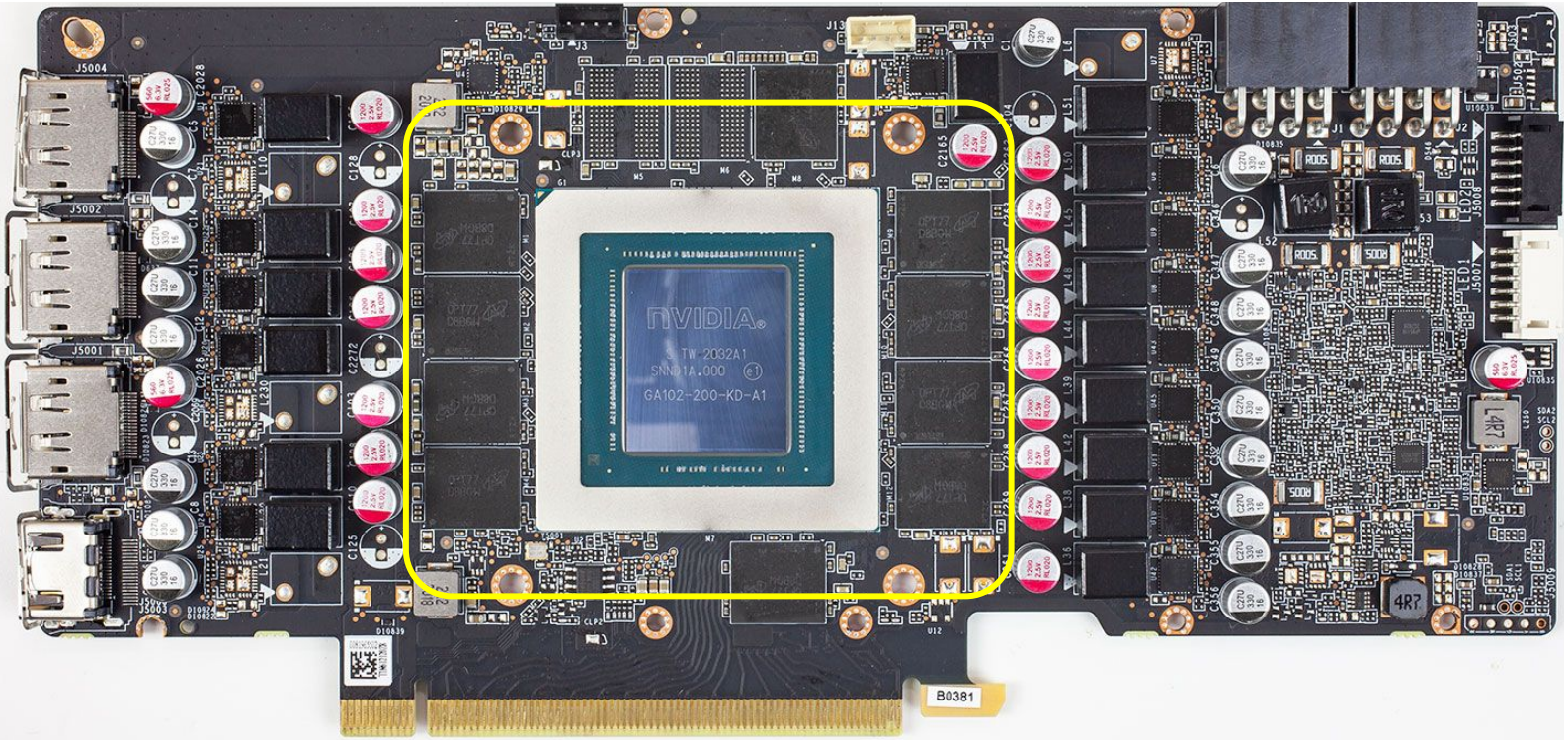
Em uma GPU não faz sentido uma cache grande.

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 i9 9900k
Cache	4 MiB	16 MiB
Largura de Banda de memória	717 GiB/s	41.6 GiB/s

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



	Nvidia QUADRO GP100	Intel i9 9900k
Cache	4 MiB	16 MiB
Largura de Banda de memória	717 GiB/s	41.6 GiB/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.

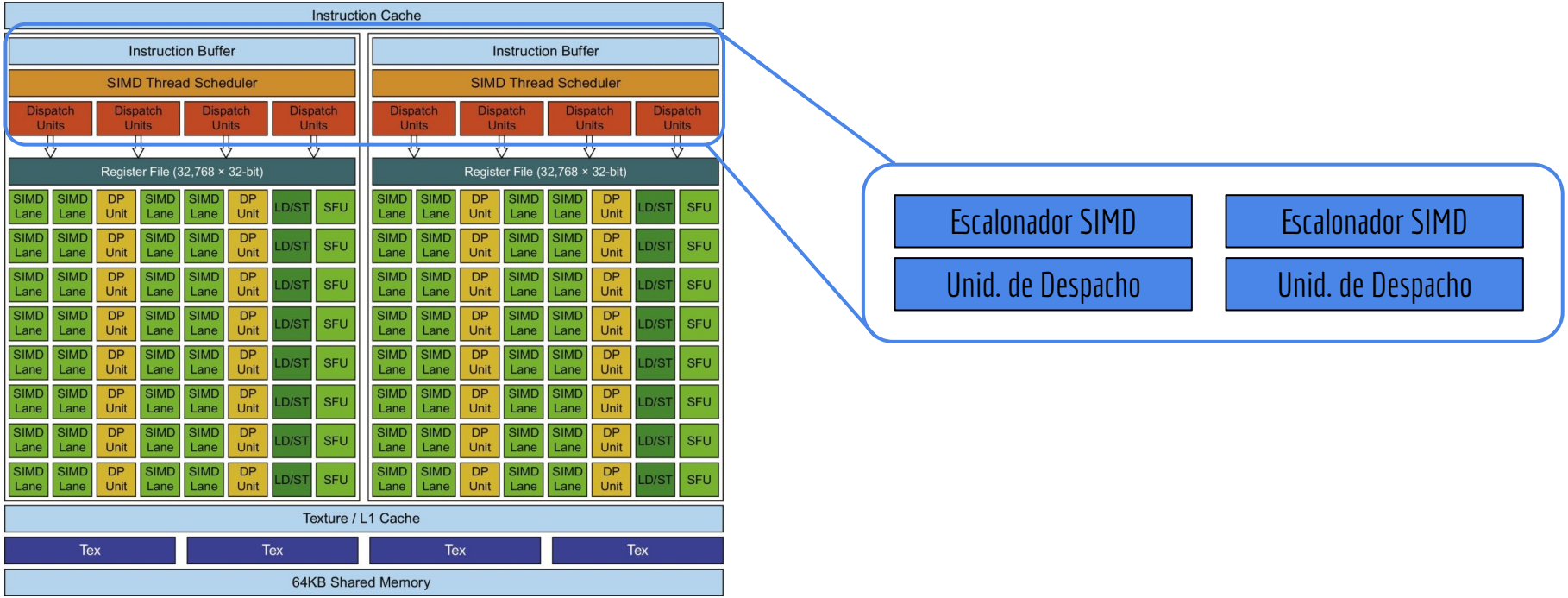
- Executa as instruções “prontas”, mesmo fora de ordem.
  - Ideia parecida com a explorada no algoritmo de Tomasulo.

Não garantimos a ordem de execução.

- Somente que tudo será executado.

Agora os MSPs não ficam parados, desde que hajam instruções prontas.

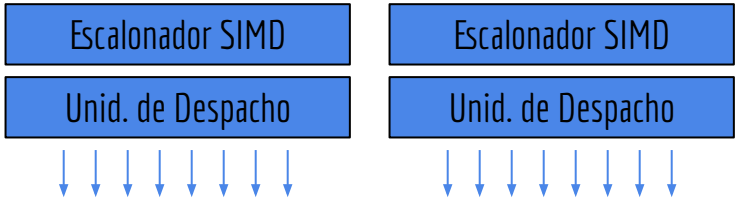
Exemplo


Instruction Cache
Instruction Buffer
SIMD Thread Scheduler
Dispatch Units
Register File (32,768 × 32-bit)
SIMD Lane
DP Unit
LD/ST
SFU
Texture / L1 Cache
Tex
64KB Shared Memory
Escalonador SIMD
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
```

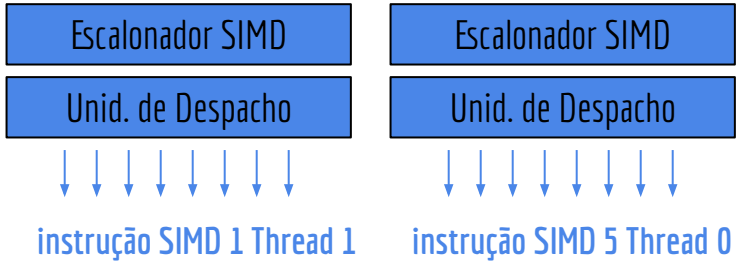


Não pronto
Pronto.
Pronto e em execução.
Pronto e executado.

Exemplo

Thread Block 0

...	...
não pronta	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
não pronta	instrução SIMD 2 Thread 0
pronta	**instrução SIMD 1 Thread 0**
não pronta	instrução SIMD 0 Thread 0

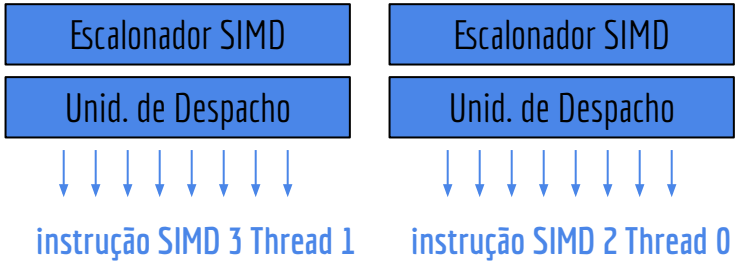


Não pronto
Pronto.
Pronto e em execução.
Pronto e executado.

Exemplo

Thread Block 0

...	...
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

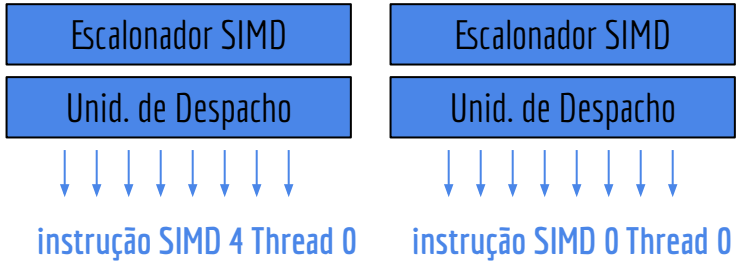


Não pronto
Pronto.
Pronto e em execução.
Pronto e executado.

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

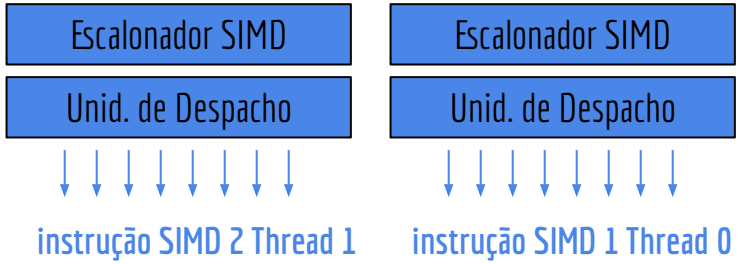


Não pronto
Pronto.
Pronto e em execução.
Pronto e executado.

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
```

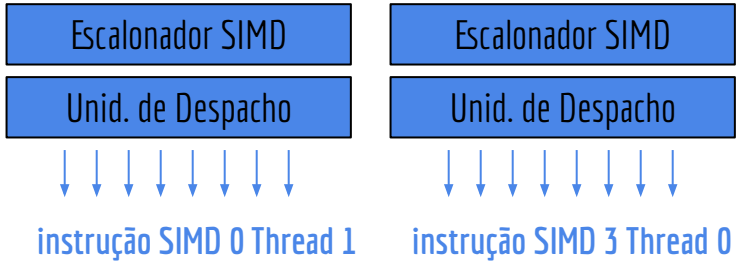


Não pronto
Pronto.
Pronto e em execução.
Pronto e executado.

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
```

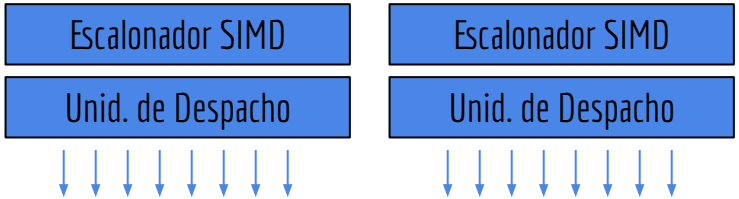


Não pronto
Pronto.
Pronto e em execução.
Pronto e executado.

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.

O desafio em se programar uma GPU

O desafio para o programador então é **criar um programa composto por múltiplas instruções independentes.**

Dessa forma o escalonador pode escolher as instruções **em qualquer ordem**.

Divergência de WARP

Uma CPU sabe fazer ifs e elses (branches).

Conta com circuitos sofisticados para:

- Prever desvios.
- Reduzir o custo caso a previsão de desvio cometa um erro.

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.

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

Uma GPU **não** sabe fazer ifs.

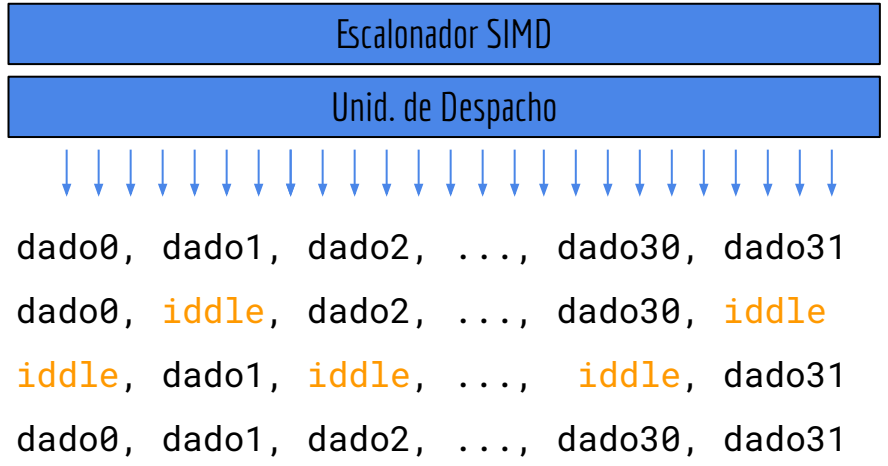
Em cada fluxo de instruções SIMD – chamado de **WARP**, as instruções devem ser executadas em sequência.

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

Resumo da ópera:

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 CUDA

Primeiro 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];
}
```

Em 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>>>();
```

Em 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.

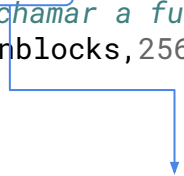
Considerando que cada bloco de threads vai operar em 256 elementos.

Em 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>>>();
```



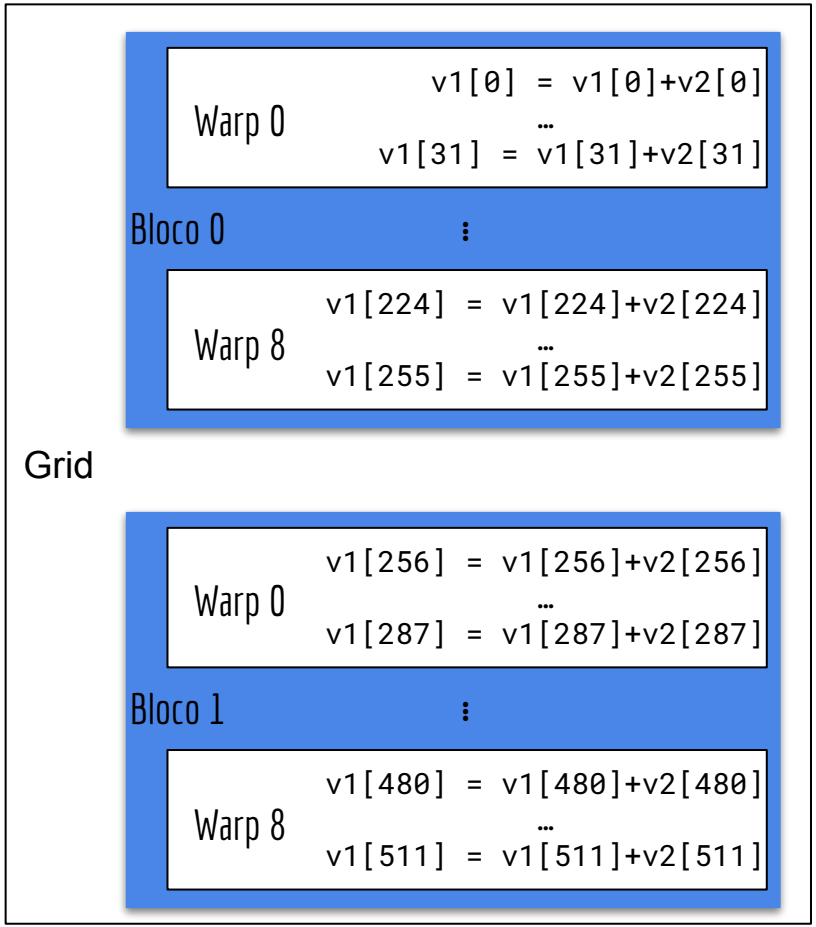
nblocks = (512 + 255) / 256 = 2 blocos.

Em 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>>>();
```

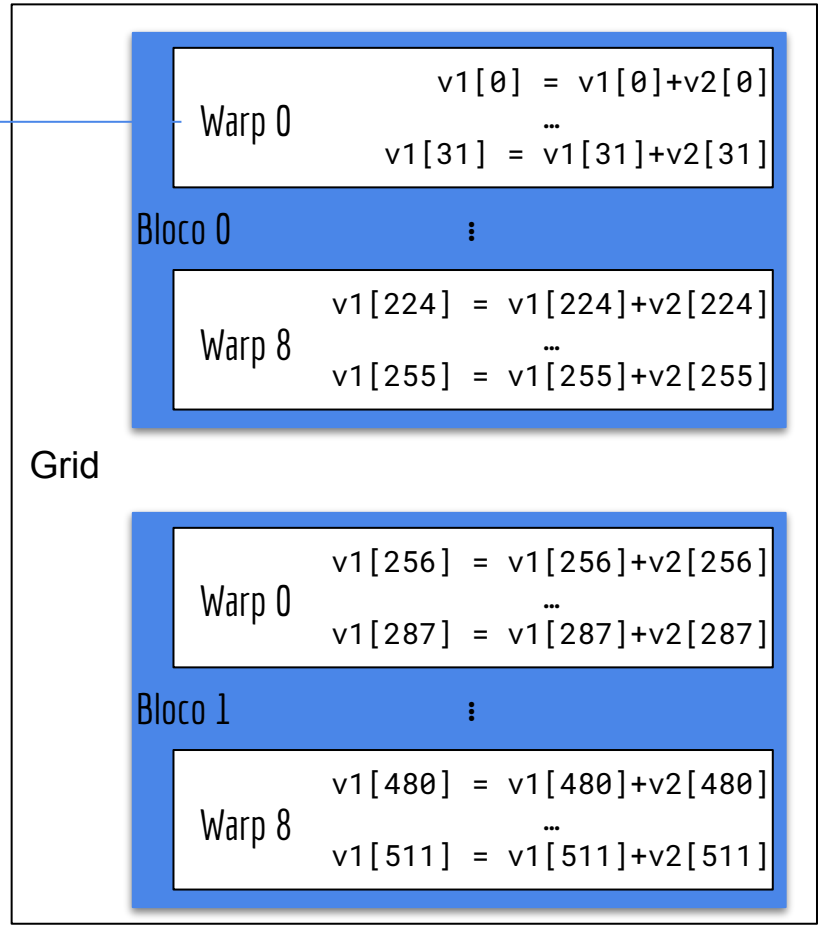


Em CUDA

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!

- Primeiro entenda a arquitetura.
- Difere de muitas formas de um programa para CPU.
  - Se torna mais claro quando você entende as peculiaridades da GPU.

Programas em GPU estão na moda!

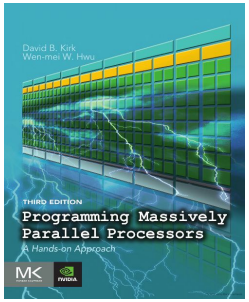
- Executar bibliotecas/funções prontas que implementam programas dentro das nossas GPUs é fácil.
- Encontrar alguém que realmente saiba criar um programa do zero para uma GPU é difícil.
  - **E esse alguém custa muito caro!**

Exercícios

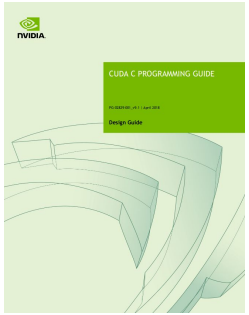
1. Pesquise no site da Nvidia as configurações de uma placa de vídeo de sua preferência. Verifique.
    O modelo do processador.
    Quantos “Cuda Cores” essa placa possui.
    Quantos cores essa placa realmente possui.
        Você pode assumir que as instruções SIMD operam em 32 dados como na arquitetura 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.

Referências

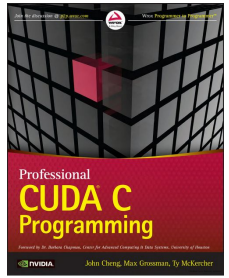
Kirk, Hwu. Programming Massively Parallel Processors: A Hands-on Approach. 3a edição. 2016.



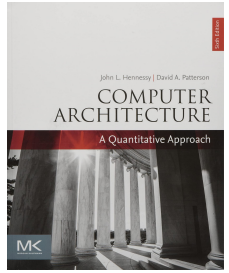
Nvidia. CUDA C Programming Guide. 2018.



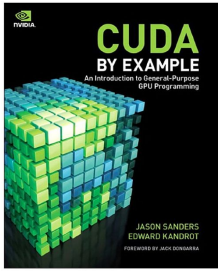
Cheng; Grossman; McKercher. Professional CUDA C Programming. 2014.



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



Sanders; Kandrot. CUDA by Example. 2010.



Licença