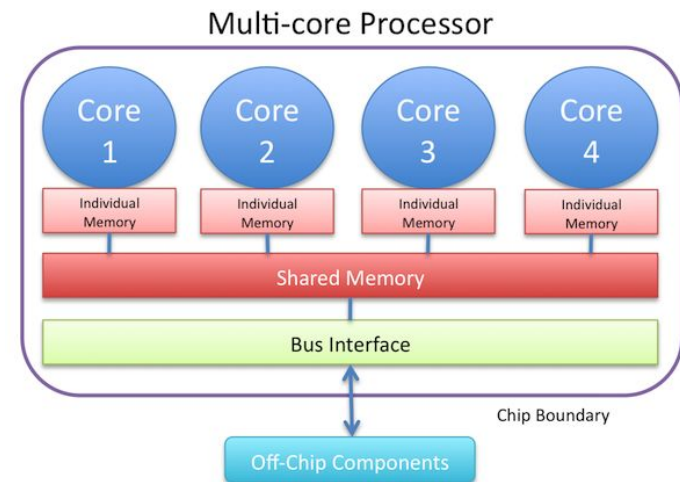


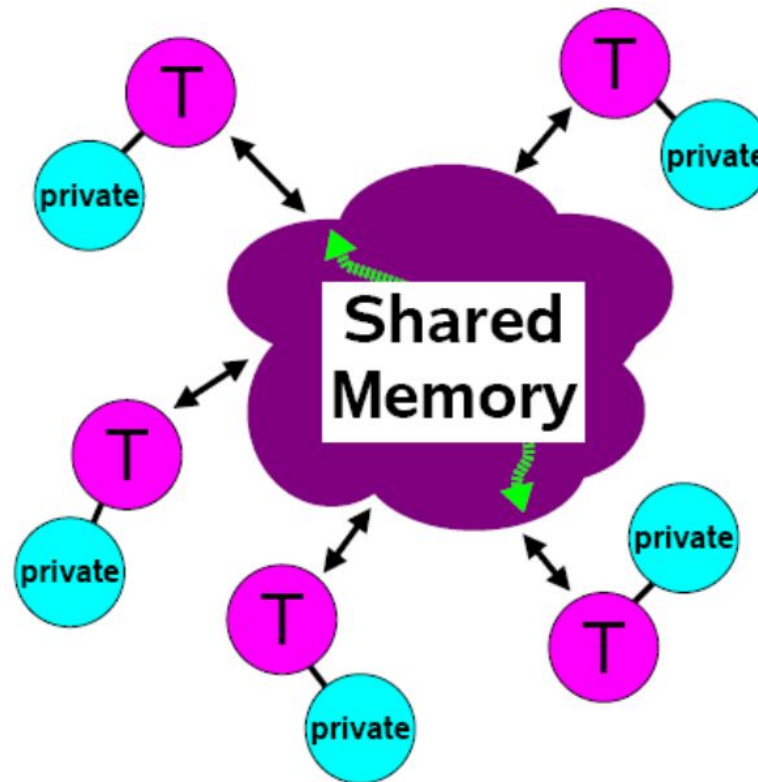
Multicore+OpenMp

Processadores Multicore

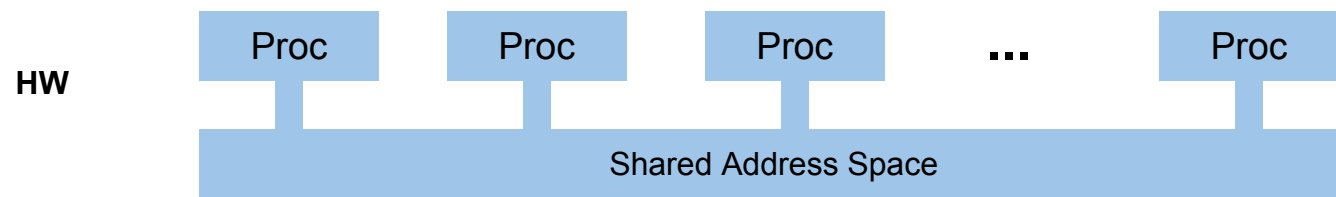
- Porque surgiram?
 - Diminuir o consumo de energia
 - Aumentar a capacidade de processamento sem aumentar a liberação de calor no chip
 - Vários cores de processamento em único chip



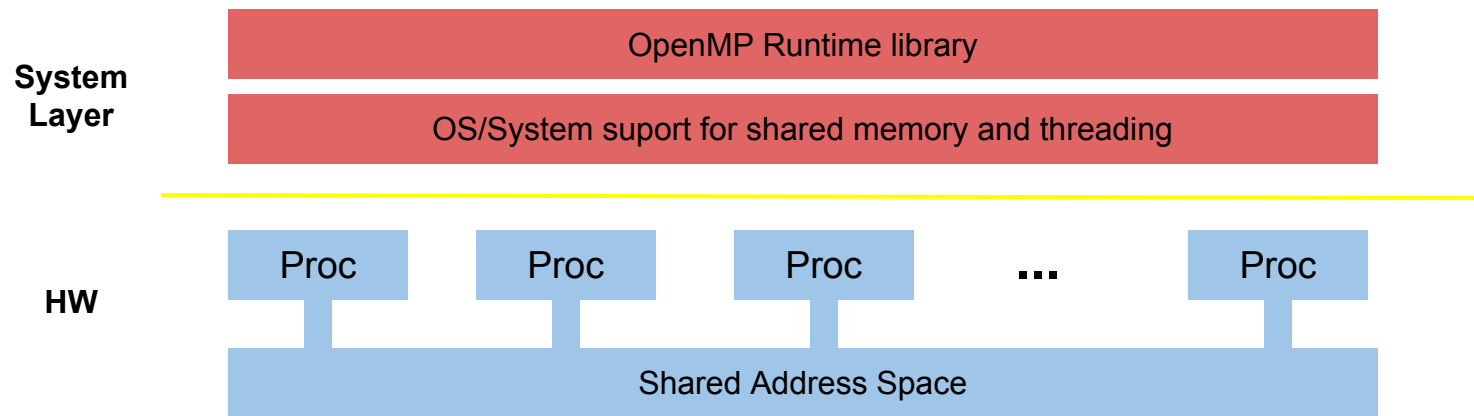
Memória Compartilhada



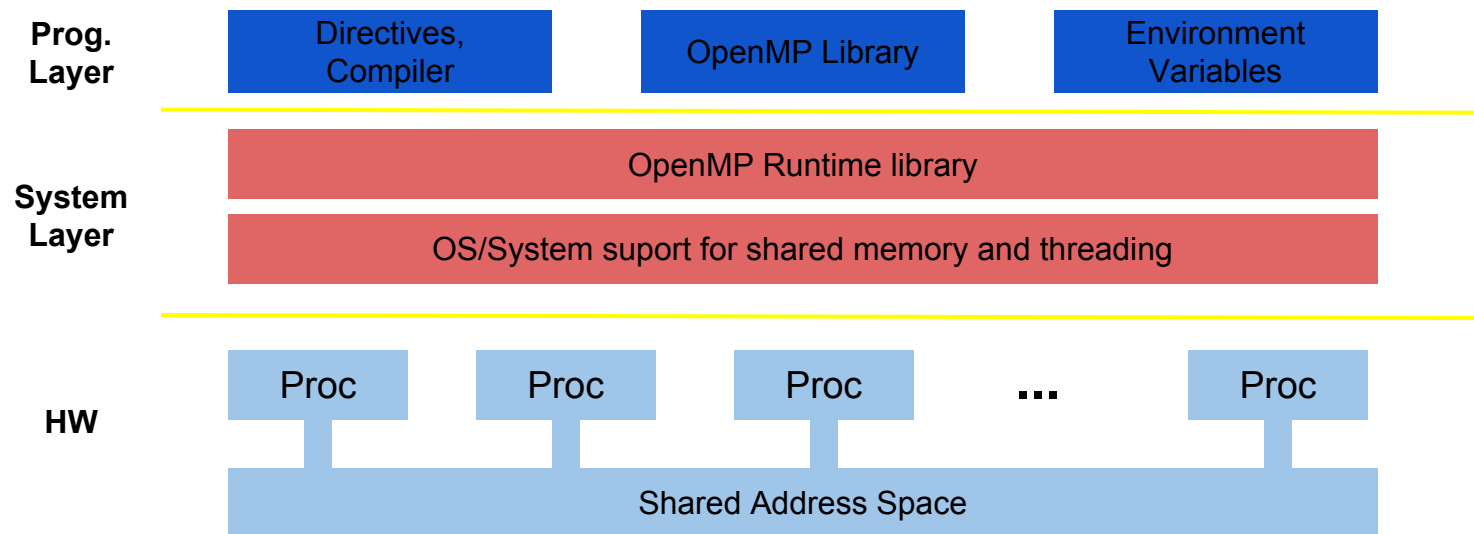
OpenMP



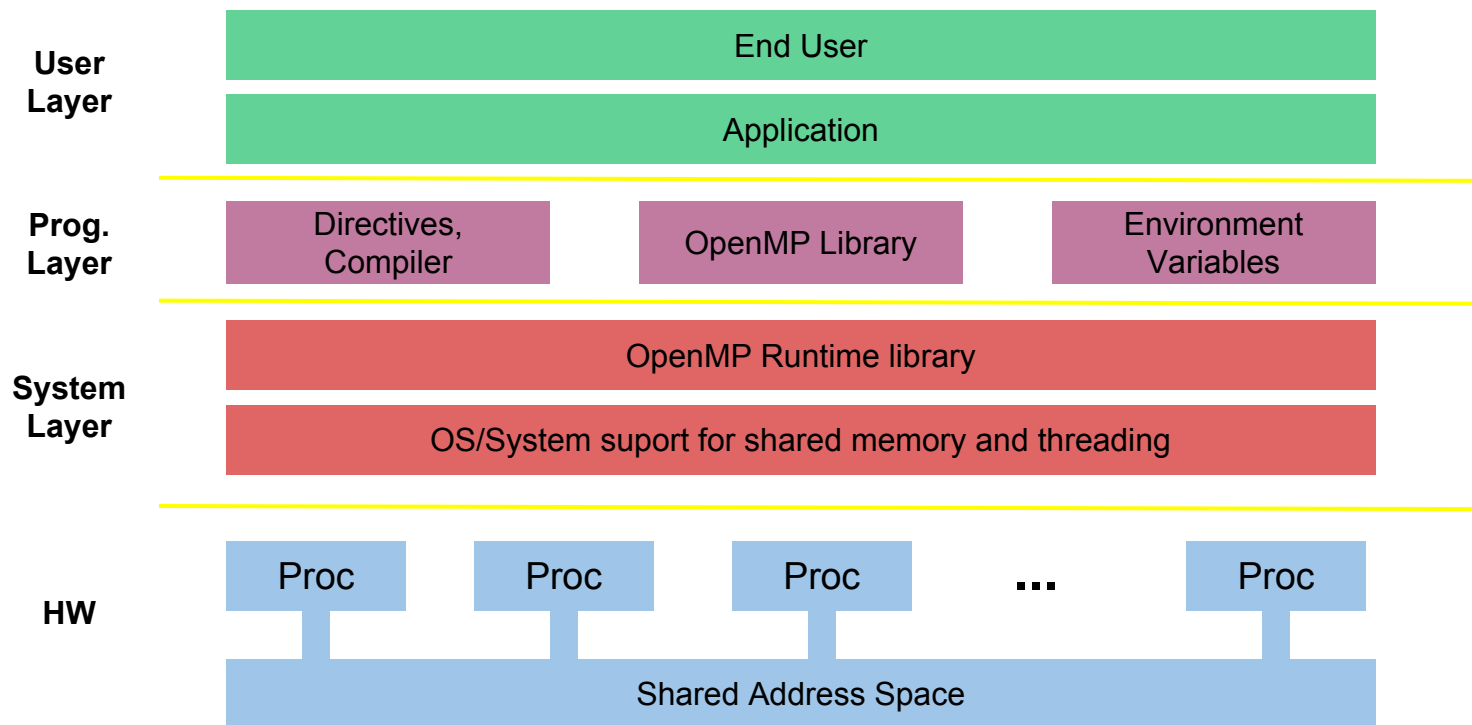
OpenMP



OpenMP



OpenMP

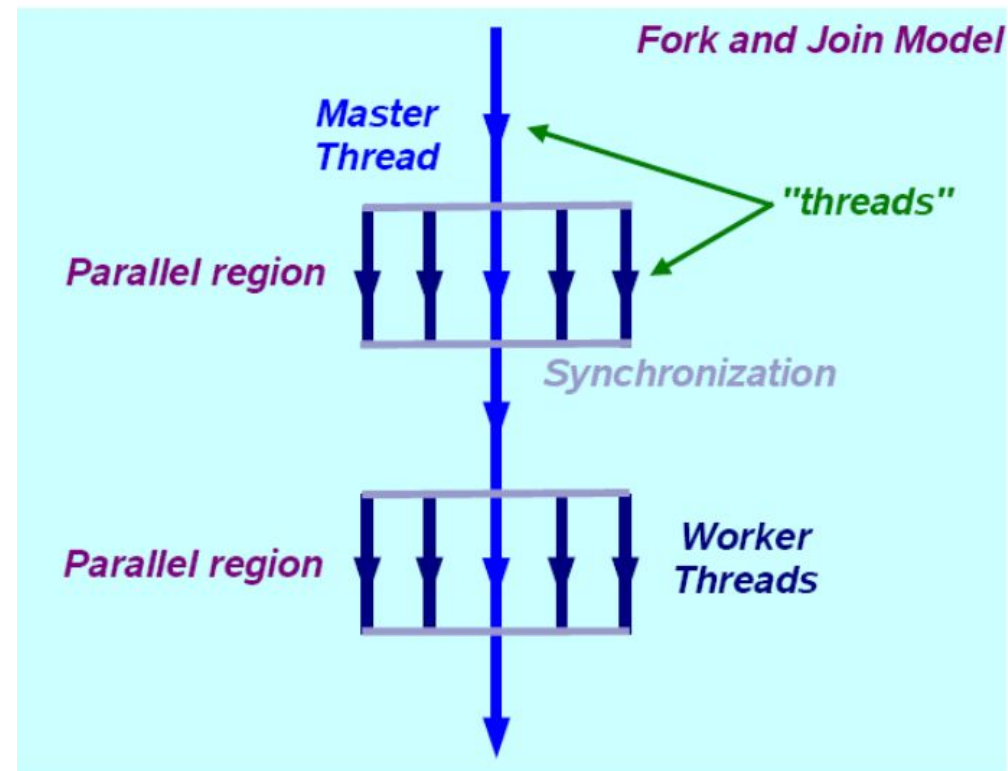


OpenMP

- Ferramenta de programação para memória compartilhada.
- Modelo de programação fork-join, com geração de múltiplas threads.
- Inicialmente executa-se um thread até que aparece o primeiro construtor paralelo, e criam-se as outras threads.
- Ao final do construtor se sincronizam as threads e continuam a execução.



Modelo Fork-Join



OpenMP

- OpenMP é formado por:
 - Construtores: indicam como distribuir o trabalho, gerenciar as threads e como sincronizar.
 - Funções: para estabelecer, obter e comprovar valores.
 - Variáveis de ambiente: indicam a forma da execução.



Construtores (pragma)

Sintaxe:

```
#pragma omp nome [cláusulas]
```




Número de threads

- Determinado pelos seguintes fatores:
 - Função **omp_set_num_threads()**
 - Variável de ambiente `OMP_NUM_THREADS`
 - Implementação padrão do ambiente: número de processadores em um nó
-

OpenMP

```
#include <omp.h>
int main(int argc, char **argv){
    int nthreads, tid;
    #pragma omp parallel private(nthreads, tid)
    { // Obtém e imprime o id do Thread
        tid = omp_get_thread_num();
        printf("Alo da thread = %d\n", tid);
        if (tid == 0) // apenas o master thread faz isto
        {
            nthreads = omp_get_num_threads();
            printf("Msg Master: Existem + %d\n", nthreads);
        }
    } //Sincronismo de todos os threads
    return 0;
}
```



Construtor Paralelo

```
#pragma omp parallel [atributo...] nova linha
    if (expressão lógica)
    private (lista)
    shared (lista)
    default (shared | none)
    firstprivate (lista)
    reduction (operador:lista)
    copyin (lista)
```

estrutura de blocos

OpenMP

Exemplo:

Hello World



Construção de Trabalho Compartilhado

```
#pragma omp for [atributo...] nova linha
    schedule (tipo [,chunk])
    ordered
    private (lista)
    firstprivate (lista)
    lastprivate (lista)
    shared (lista)
    reduction (operador:lista)
    nowait
for_loop
```

OpenMP

Exemplo:

Soma de Vetores



OpenMP

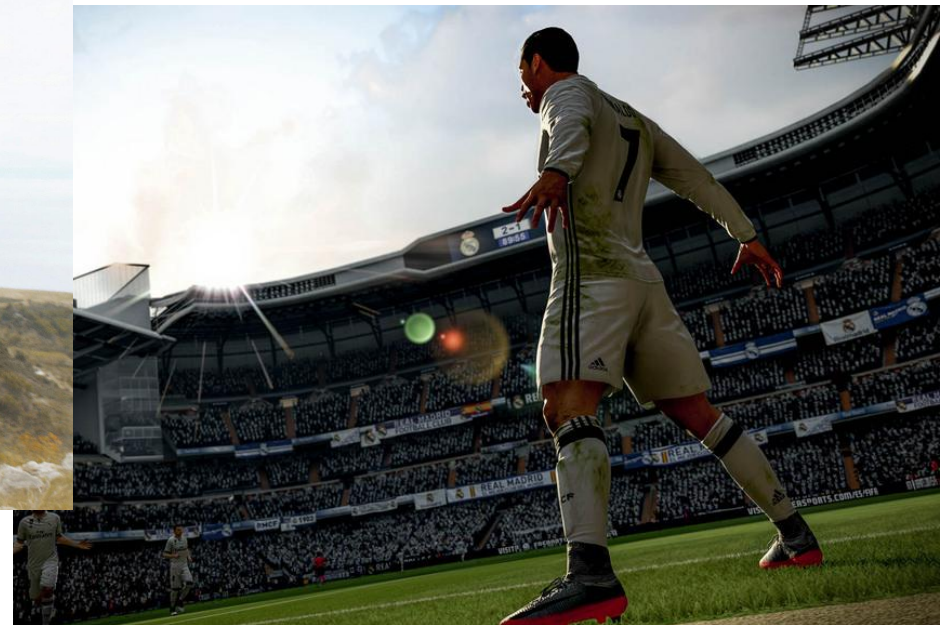
Exemplo:

Pareamento Probabilístico de Registros (Tempo de Execução)



GPU+CUDA

Desafios da Computação Gráfica

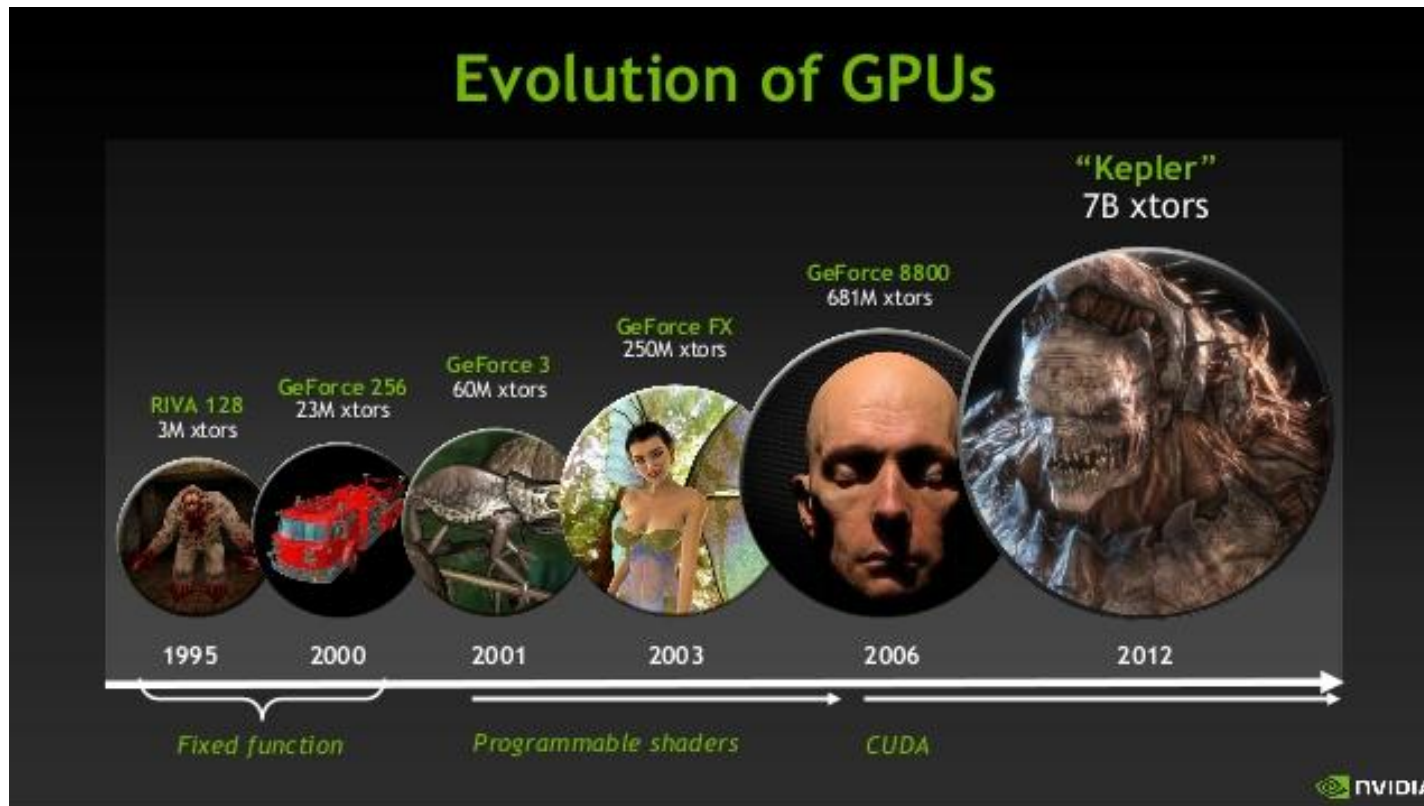


Graphics Processing Unit (GPUs)

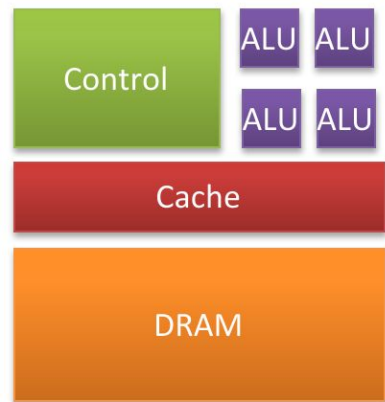
- O processamento gráfico caracteriza-se por uma computação massiva.
- O processamento gráfico é intrinsecamente muito paralelizável.
 - Muitas operações são feitas através da aplicação de um paralelismo com granularidade fina



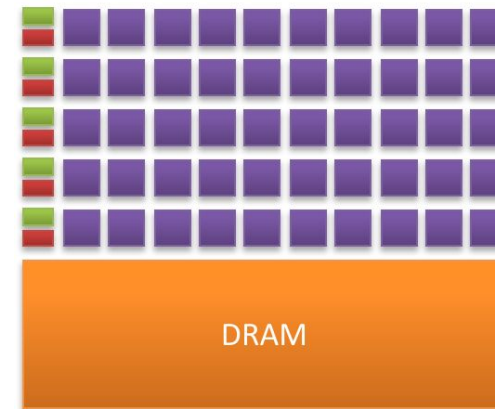
Graphics Processing Unit (GPUs)



Arquitetura das CPUs vs GPUs

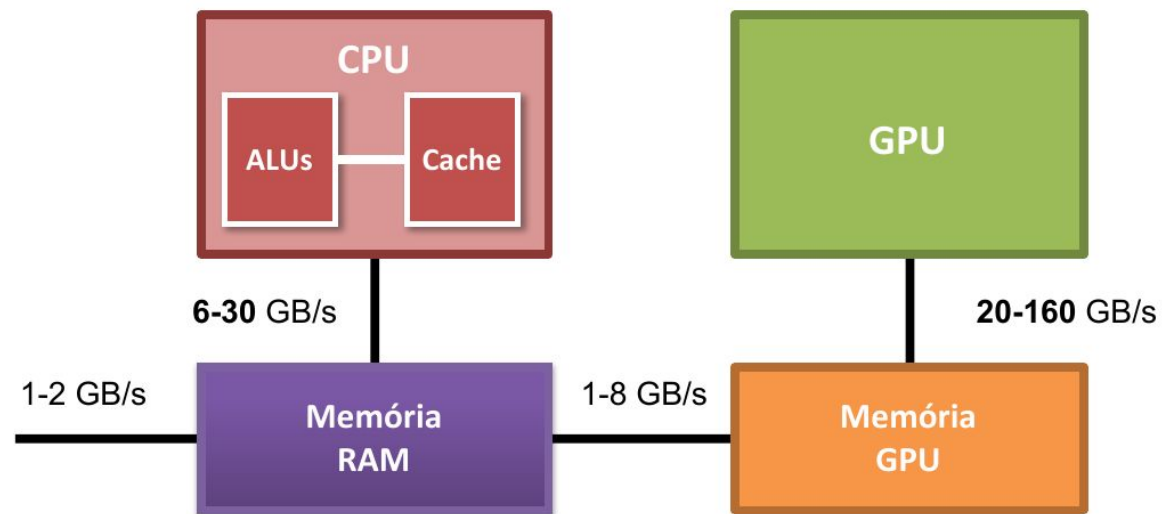


- Grandes **caches**
- Poucos elementos de processo
- Otimização das Localidades Espacial e Temporal
- Sofisticados sistemas de controle



- Caches reduzidas
- Elevado número de elementos de processo
- Acesso otimizado para acesso sequencial de dados (streaming)

Largura de Banda



Arquitetura CUDA



Como programar GPUs?

- CUDA Extensão do C, C++, Fortran
- Objetivo
 - Escalabilidade em 100's de cores, 1000's threads paralelas
 - Foco no desenvolvimento de algoritmos paralelos
 - Possibilita computação heterogênea (CPU+GPU)
- Define
 - Modelo de Programação
 - Arquitetura



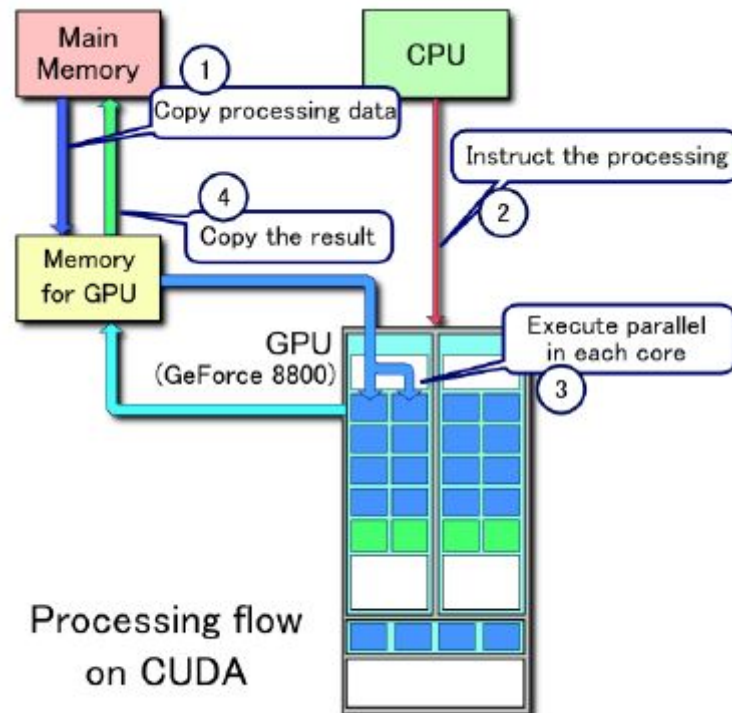
Como programar GPUs?

Exemplo:

Hello World!

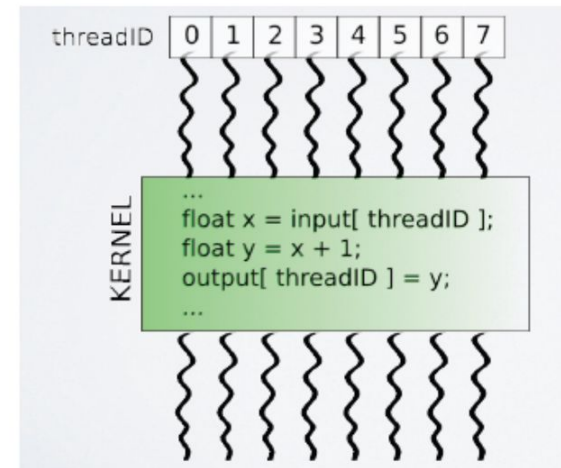


Fluxo de Execução



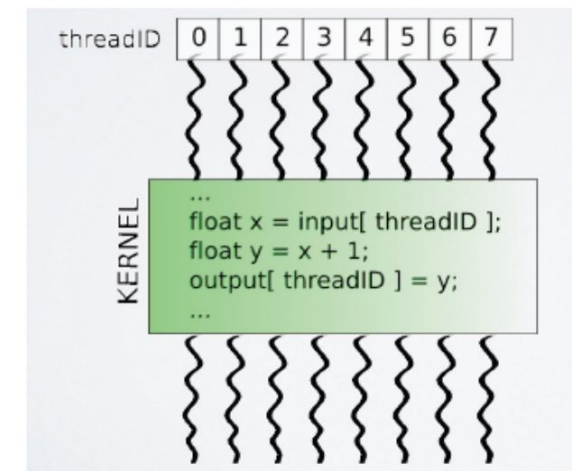
Modelo de Execução

- Modelo de Programação SIMD.
- A unidade de execução chama-se **kernel**
 - O paralelismo se baseia em threads.
 - O kernel de execução consistirá em um conjunto de threads.



Modelo de Execução

- Todos os threads executam o mesmo código
- Cada thread recebe um ID único
 - Em função do ID: toma-se decisões de controle de fluxo ou fará operações sobre os dados
 - Identificadores intrínsecas: threadIdx, blockIdx, blockDim, gridDim



Executando kernels

- Chamada de função modificada:

```
kernel<<<dim3 grid, dim3 block>>> (...)
```

- Exemplo:

```
dim3 grid(16, 16)  
dim3 block(16, 16)  
kernel<<<grid, block>>> (...)  
  
kernel<<<32, 512>>> (...)
```

Executando kernels

```
__global__ void kernel(int *dev_a, int value){  
    int idx = blockDim.x * blockIdx.x + threadIdx.x  
    dev_a[idx] = value;  
}
```


Executando kernels

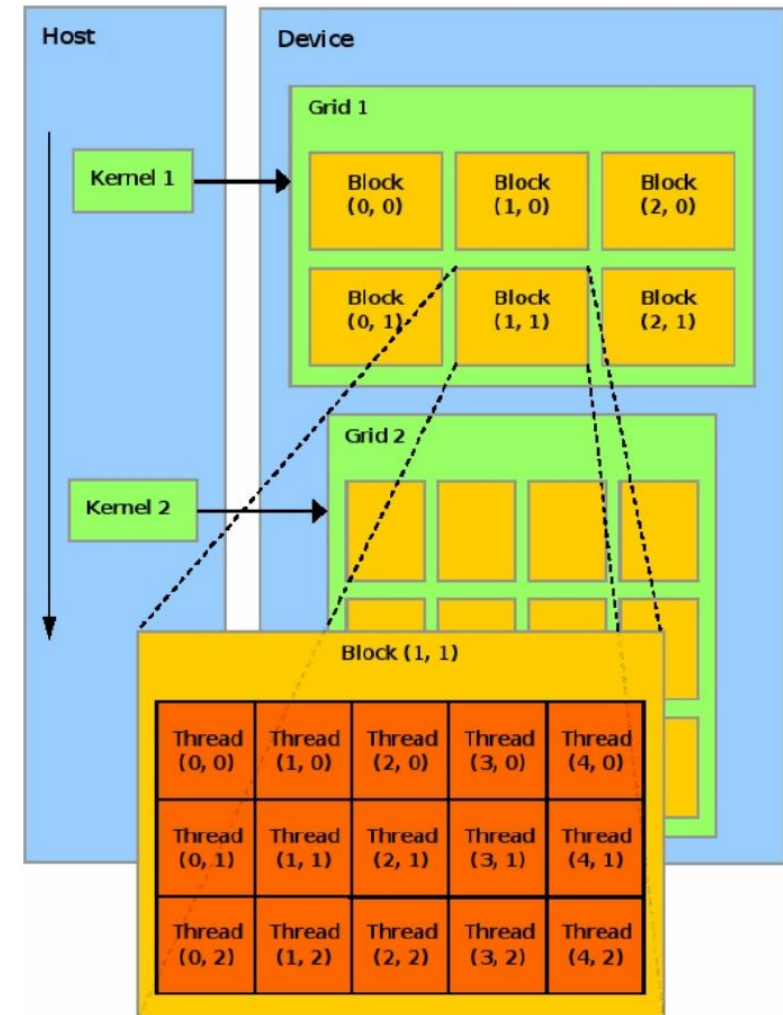
Exemplo:

Soma de dois elementos na GPU.



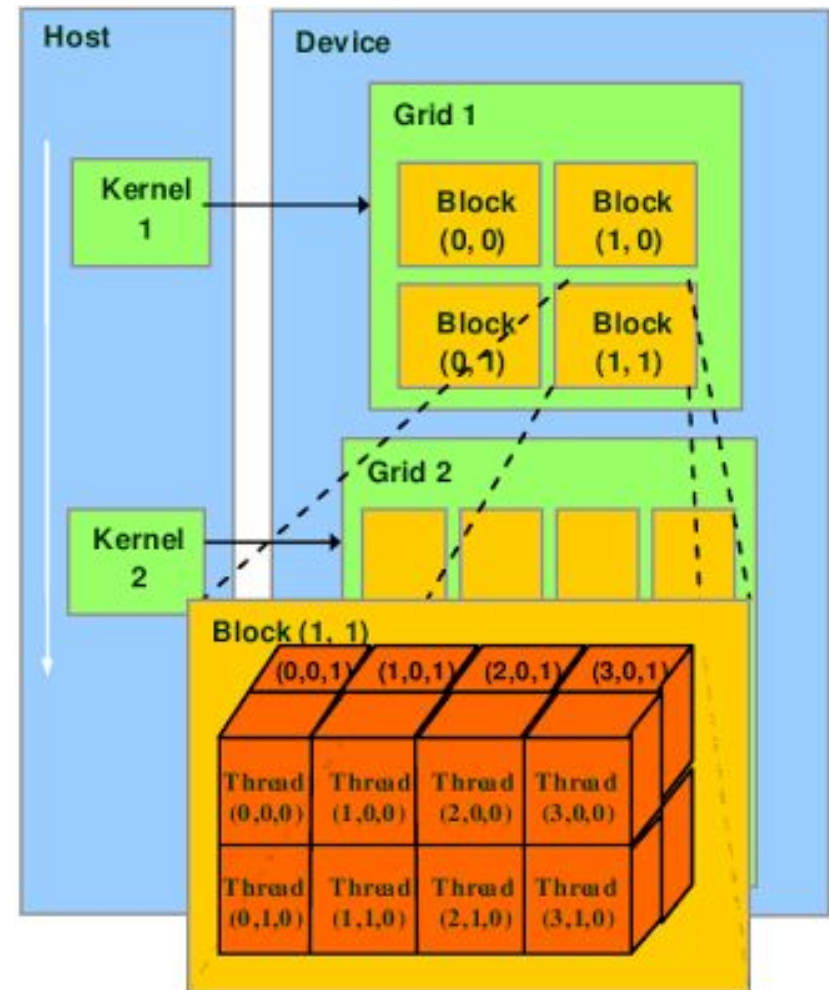
Modelo de Programação

- Os threads de um kernel constituem um grid.
- Os threads em um grid se agrupam em blocos.
 - Sincronizam sua execução
 - Comunicam-se através da memória compartilhada.



Modelo de Programação

- Cada thread tem associado:
 - Identificador próprio dentro do bloco (1D, 2D o 3D)
 - Identificador do bloco que pertencem dentro do grid
- Um warp é um bloco de threads em execução
 - Cada warp consiste em 32 threads



Transferências de dados

- Chamada desde o Host
 - `cudaMemcpy(void * dst, void * src, size_t nbytes, enum cudaMemcpyKind direction)`
 - API
 - `cudaMemcpyHostToDevice`
 - `cudaMemcpyDeviceToHost`
 - `cudaMemcpyDeviceToDevice`
-

Gestão da Memória

- O host (CPU) realiza a reserva e liberação de memória
 - `cudaMalloc(void ** ptr, size_t nbytes)`
 - `cudaMemset(void * ptr, int value, size_t count)`
 - `cudaFree(void * ptr)`

```
int main(int argc, char **argv){
    int n = 1024 * sizeof( int );
    int * d_A = NULL;

    cudaMalloc( (void **)&d_A, n );

    cudaMemset( d_A, 0, n );

    cudaFree( d_A );
}
```

Espaços em Memória

- CPU e GPU possuem espaços de memória independentes
 - Os dados se movem através do 'bus PCIeExpress'
 - Reserva/Cópia/Liberação explícitas
 - Qualquer operação Reserva/Cópia/Liberação é realizada pelo host



Executando kernels

Exemplo:

Soma de vetores

