Introdução à Programação em CUDA

Pedro Lara

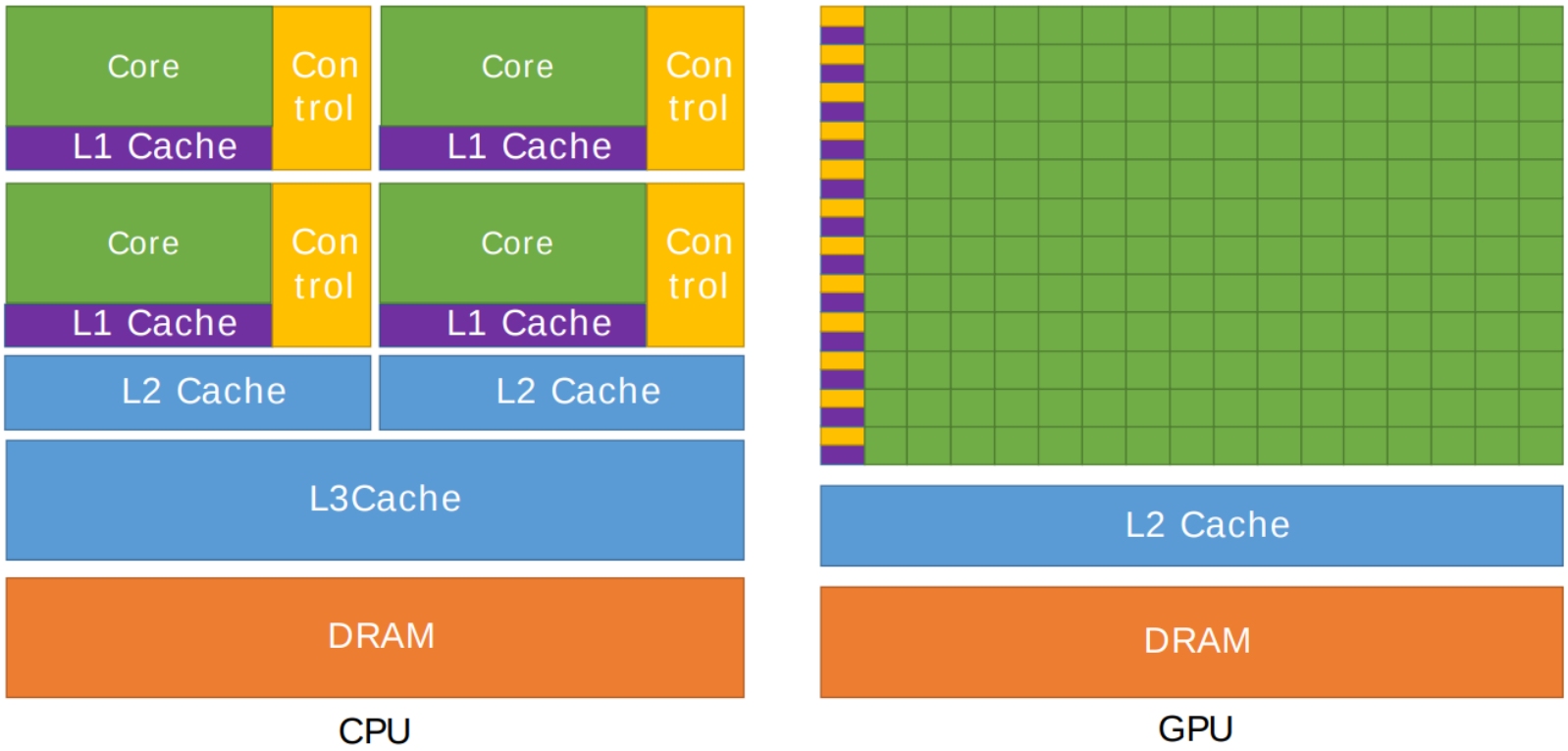
[1]CEFET-RJ

14 de Outubro de 2020

O que é CUDA?

Compute Unified Device Architecture (inicialmente proposto) é uma arquitetura, associada majoritariamente a dispositivos NVIDIA, para computação paralela em GPGPU e computação heterogênea.

Por que CUDA?



Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Host e Device

- **Device:** GPGPU + sua memória
- **Host:** CPU + sua memória

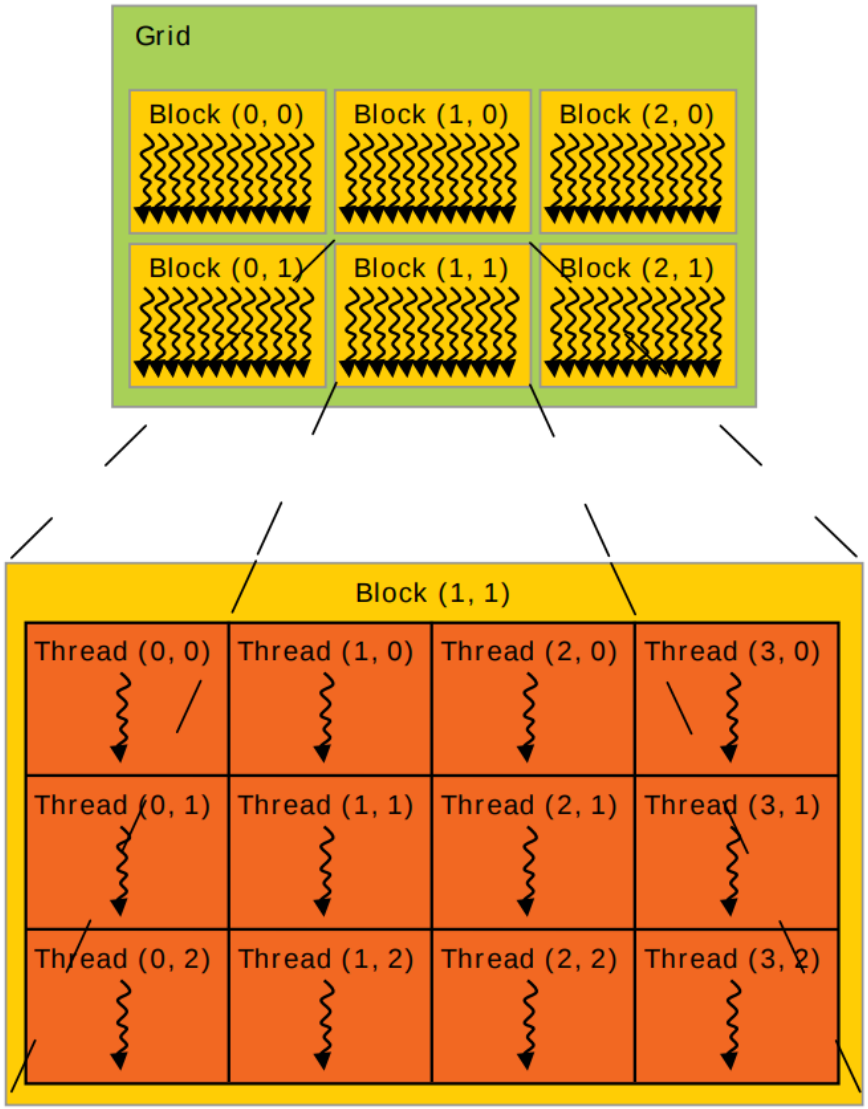
Kernel

Modelo de Programação

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

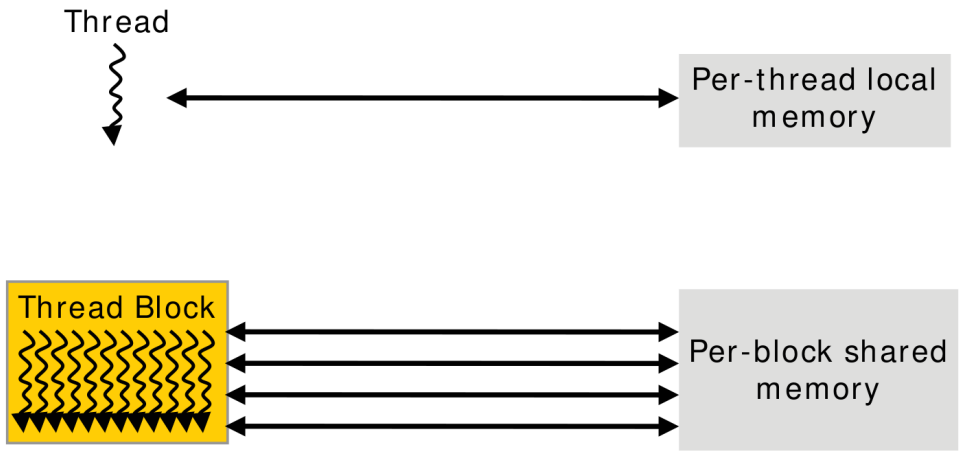
Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Grid e Block



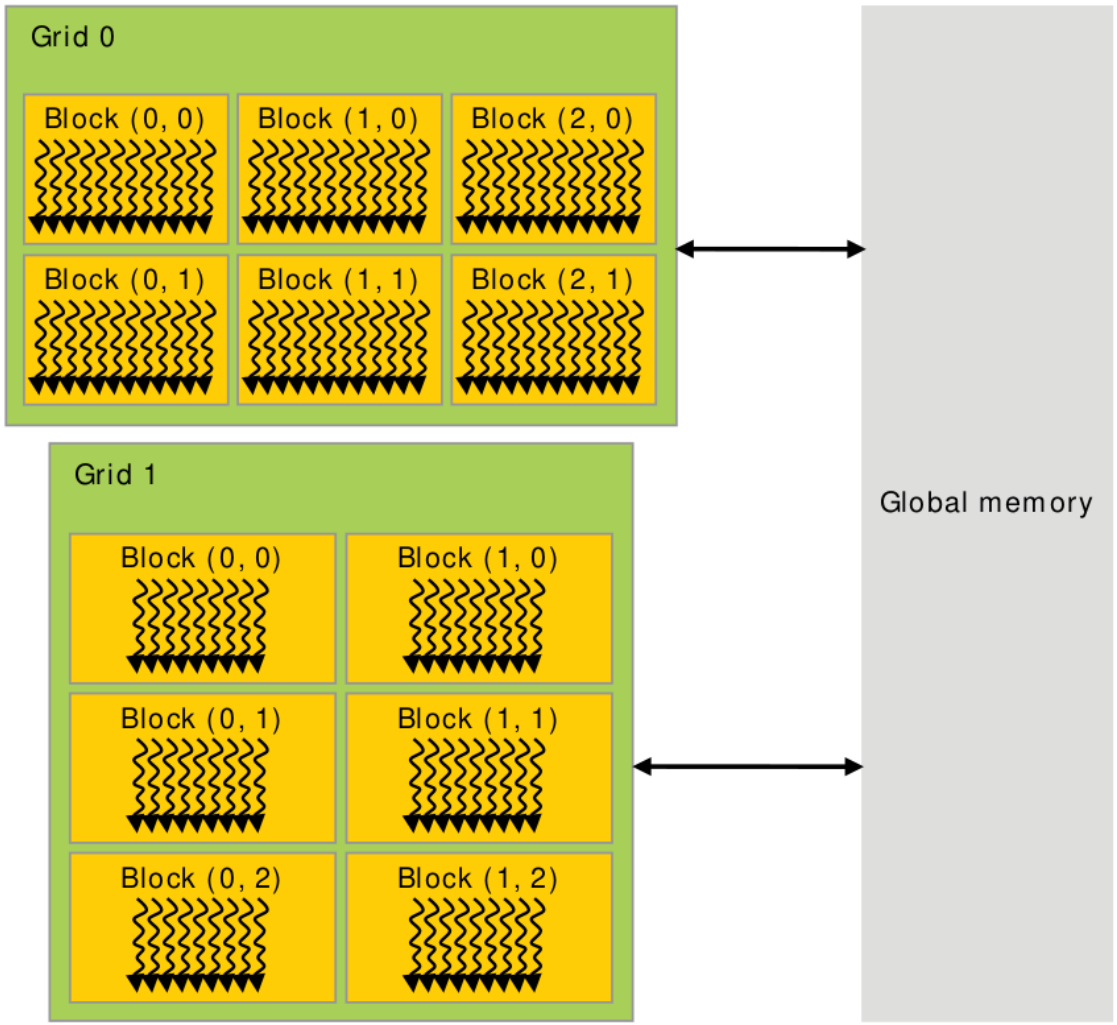
Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Hierarquia de Memória



Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Hierarquia de Memória



Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

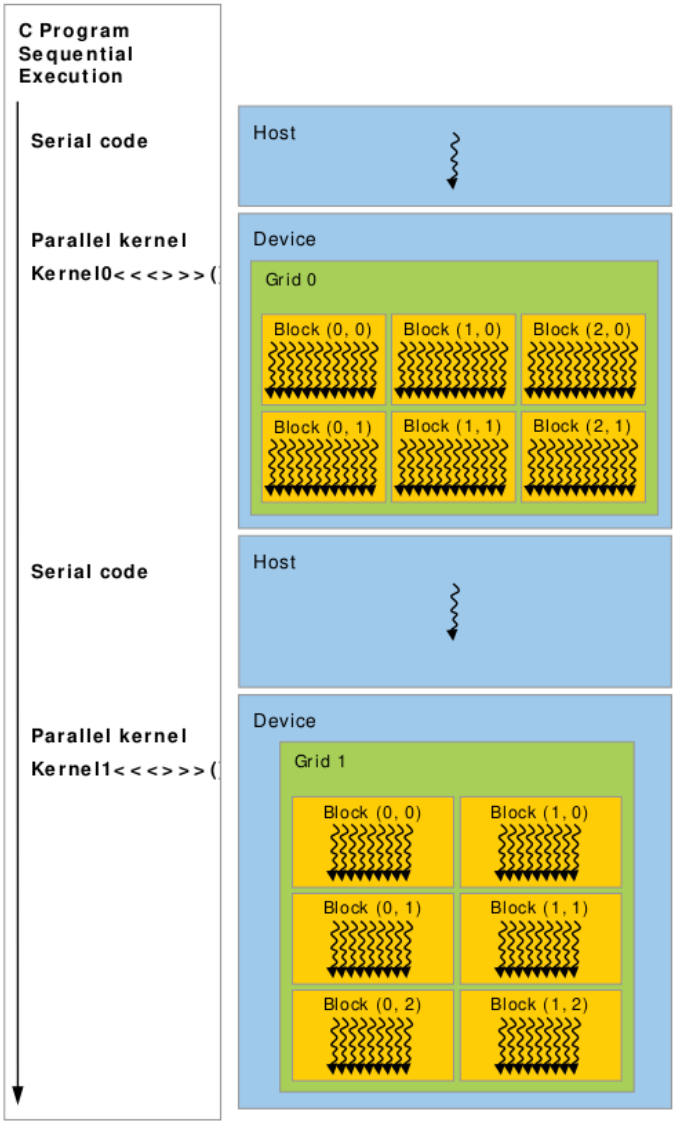
Hierarquia de Memória

As *threads* possuem três espaços de memória:

- **Local memory** Restrita à cada thread.
- **Shared memory** Compartilhada com todos os threads de um bloco.
- **Global memory** Compartilhada com todos os threads. Mesmo aqueles de execuções de kernels diferentes.

Além desses, existem também as memórias de apenas leitura: **constant** e **texture**.

Computação Heterogênea



Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

SIMT

- **SIMT** Single Instruction Multiple Thread
- Conceito semelhante ao SIMD (Single Instruction Multiple Data)
- SIMT: SIMD com múltiplas threads.

Resumindo...

- **Device:** GPGPU + sua memória.
- **Host:** CPU + sua memória.
- **Kernel:** Rotina que será executada em paralelo.
- **Thread:** Instância de um Kernel executado por um core.
- **Bloco:** Grupo de threads executado por um SM (Max. 1024).
- **Warp:** Grupo de 32 threads em um bloco que executa a mesma instrução.
- **Grid:** Grupo de blocos executado por um Device.

Índices e Dimensões

- `blockDim.x,y,z` retorna o número de threads, em um bloco.
- `gridDim.x,y,z` retorna o número de blocos, em um grid.
- `blockIdx.x,y,z` retorna o índice do blocos, em um grid.
- `threadIdx.x,y,z` retorna o índice de uma thread, em um bloco.

Índices e Dimensões

Kernel

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
```

Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Índices e Dimensões

Main

```
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Transferência de Memória entre Host e Device

```
cudaMalloc ( void** devPtr, size_t size )
```

Aloca memória em um device.

```
cudaMemcpy ( void* dst,
             const void* src,
             size_t count,
             cudaMemcpyKind kind )
```

Copia dados entre o host e device.

`kind` **pode ser** `cudaMemcpyHostToDevice` ou `cudaMemcpyDeviceToHost`;

Transferência de Memória

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Transferência de Memória

Kernel

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Detalhes Importantes

- Computação Heterogênea
- Modelo SIMT (SIMD)
- Use muitas threads
- Otimize o tráfego de memória entre host e device

Exemplo completo: VecAdd

Principais passos para o programa VecAdd

- Alocar vetores no Device.
- Copiar os dados para o Device.
- Instanciar o kernel.
- Copiar o resultado para o Host.

Exemplo: Julia Set

Fractal Conjunto de Julia é gerado mapeando cada ponto (x, y) da imagem no complexo $z_0 = x^* + iy^*$ e executando a iteração

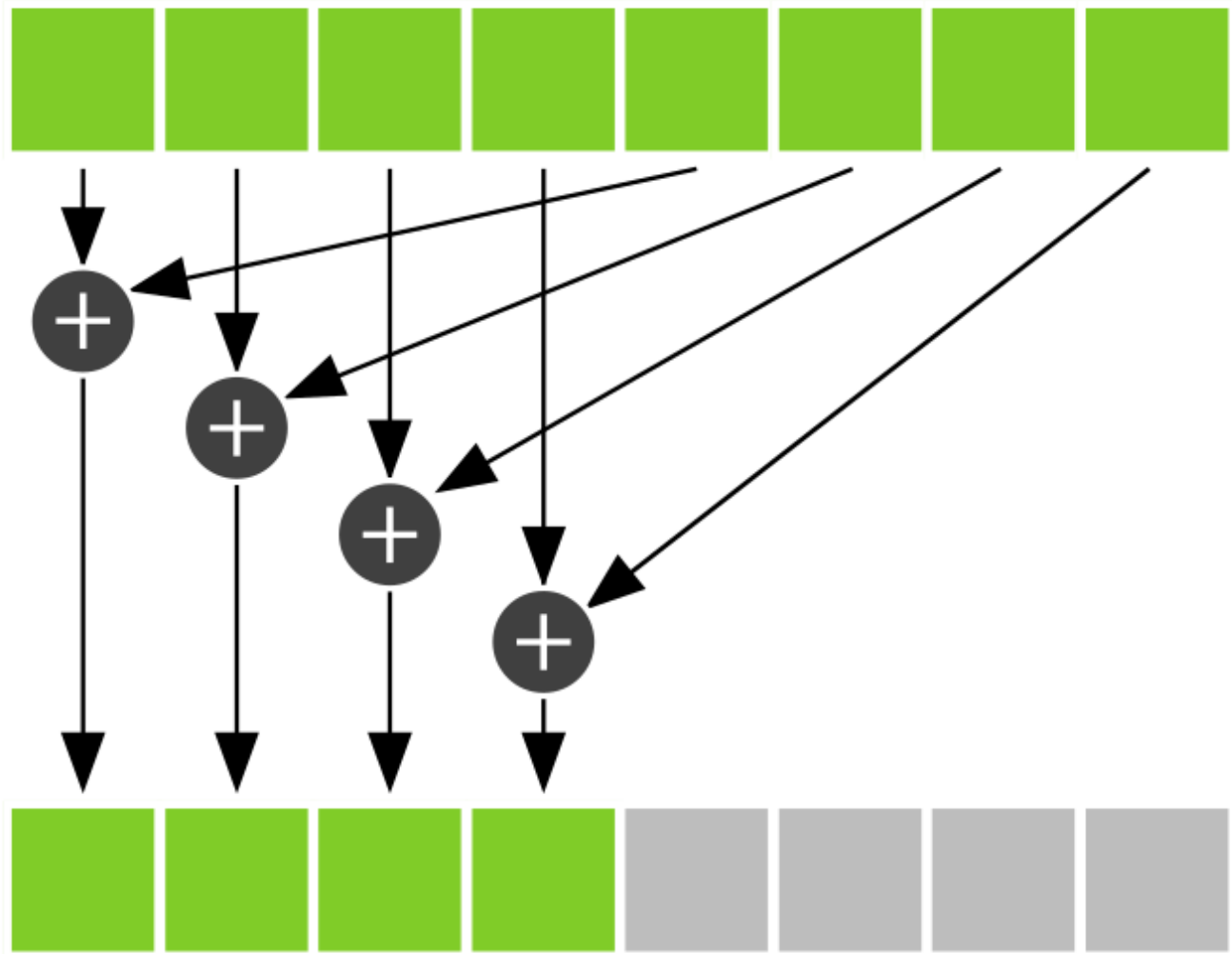
$$z_{(n+1)} = z^2_{(n)} + c$$

Até que z_n ou n atinja um limiar. O número de iterações determina a cor do pixel (x, y)

Memória Shared

- Uso da palavra chave ___shared___ para declarar uma varíável nesta região.
- Mais rápida que a memória global. *"Shared memory buffers reside physically on the GPU as opposed to residing in off-chip DRAM." NVIDIA*
- Cada bloco possui uma região shared e não é possível um bloco ler/modificar dados desta região de outro bloco.

Memória Shared Exemplo Redução



Fonte: CUDA C++ Programming Guide, NVIDIA, 2021

Memória Shared Exemplo Redução

Estratégia:

- Criar um espaço de __shared__ do tamanho de threads por bloco.
- Copiar cada valor para esse espaço.
- Executar uma iteração e chamar o __syncthreads() para não criar uma condição de corrida na próxima chamada.

Memória Shared Exemplo Redução

Calcular

$$\sum_{i=0}^{N-1} x[i]$$

```
 2 __global__ void vector_sum( float * x, float * output ) {
 3    extern __shared__ float partial_sum[];
 4    int gid =  blockIdx.x*blockDim.x+threadIdx.x;
 5    int lid =  threadIdx.x;
 6    int block_size = gridDim.x;
 7    partial_sum[lid] = x[gid];
 8    __syncthreads();
 9    for( int i = block_size/2; i > 0; i /= 2 ) {
10       if(lid < i) {
11          partial_sum[lid] += partial_sum[lid + i];
12       }
13       __syncthreads();
14    }
15    if(lid == 0) {
16       output[ blockIdx.x ] = partial_sum[0];
17    }
18 }
```

Memória Shared Exemplo Redução

Exemplo um pouco mais elaborado: Mínimos Quadrados

$$\hat{\alpha} = \bar{y} - \hat{\beta}\,\bar{x}$$

$$\hat{\beta} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} = \frac{\mathrm{cov}(X, Y)}{\sqrt{\mathrm{var}(X) \cdot \mathrm{var}(Y)}}$$

$$\bar{x} = \sum_{i=1}^{n} \frac{x_i}{n} \qquad \bar{y} = \sum_{i=1}^{n} \frac{y_i}{n}$$

Memória Constante

- Memória constante utiliza a palavra chave ___constant___
- Deve ser declarada em um escopo global (não pode ser declarada dentro do kernel)
- Tem que ser alocada de forma estática.

Memória Constante

Soma e multiplica por um fator constante.

```
3 __constant__ float factors[NBLOCKS];
4
5 __global__ void sum_and_multiply(float * A, float * B, float * C ) {
6     int gid = (blockIdx.x * blockDim.x) + threadIdx.x;
7     C[gid] = (A[gid] + B[gid]) * factors[blockIdx.x];
8 }
```

Para copiar

```
cudaMemcpyToSymbol(factors, hostArray, NBLOCKS*sizeof(float), 0, cudaMemcpyHostToDevice));
```

Profiling Cuda Codes

Medindo tempo em trechos de código

```
cudaEvent_t start, stop;
float elapsedTime;
cudaEventCreate(&start);
cudaEventRecord(start,0);

// Chamando o kernel, por exemplo ...

cudaEventCreate(&stop);
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start,stop);
printf("Elapsed time : %f ms\n" ,elapsedTime);
```

Atomics

- Algumas operações seguem o modelo *read-modify-write*
- Por exemplo, x++
- Vamos supor que 2 threads precisem executar um incremento na variável x.
- Temos que evitar uma condição de corrida.

Atomics

Duas threads incrementando x. Comportamento esperado

STEP	EXAMPLE
1. Thread A reads the value in x.	A reads 7 from x.
2. Thread A adds 1 to the value it read.	A computes 8.
3. Thread A writes the result back to x.	x <- 8.
4. Thread B reads the value in x.	B reads 8 from x.
5. Thread B adds 1 to the value it read.	B computes 9.
6. Thread B writes the result back to x.	x <- 9.

Atomics

Duas threads incrementando x. Operações intercaladas

STEP	EXAMPLE
Thread A reads the value in x.	A reads 7 from x.
Thread B reads the value in x.	B reads 7 from x.
Thread A adds 1 to the value it read.	A computes 8.
Thread B adds 1 to the value it read.	B computes 8.
Thread A writes the result back to x.	x <- 8.
Thread B writes the result back to x.	x <- 8.

Atomics

Exemplo, Histograma

2	2	1	2	1	2	2	1	1	1	2	1	1	1
A	C	D	G	H	I	M	N	O	P	R	T	U	W

Figure 9.1 Letter frequency histogram built from the string *Programming with CUDA C*

Atomics

Operações suportadas

- Addition/subtraction: atomicAdd, atomicSub
- Minimum/maximum: atomicMin, atomicMax
- Conditional increment/decrement: atomicInc, atomicDec
- Exchange/compare-and-swap: atomicExch, atomicCAS
- More types in Fermi: atomicAnd, atomicOr, atomicXor

Streams

Stream Default
Transferência: Bloqueante/síncrona
Execução Kernel: Não-bloqueante/assíncrona

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);

kernel<<<1,N>>>(d_a)

cudaMemcpy(h_a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Streams

Stream Default
Transferência: Bloqueante/síncrona
Execução Kernel: Não-bloqueante/assíncrona
Pode-se executar algo na CPU concomitantemente a GPU

```
cudaMemcpy(d_a, h_a, numBytes, cudaMemcpyHostToDevice);

kernel<<<1,N>>>(d_a)

cpuComputing();

cudaMemcpy(h_a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Streams

Criando Streams

```
cudaStream_t stream1;
cudaStreamCreate(&stream1)

...

cudaStreamDestroy(stream1)
```

Streams

Para enviar dados em um stream usando cudaMemcpyAsync()

```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
cudaMemcpyAsync(d_a, h_a, N, cudaMemcpyHostToDevice, stream1);

...

cudaMemcpyAsync(h_a, d_a, N, cudaMemcpyDeviceToHost, stream1);
cudaStreamDestroy(stream1);
```

Streams

Ao invocar o kernel podemos referenciar o stream específico

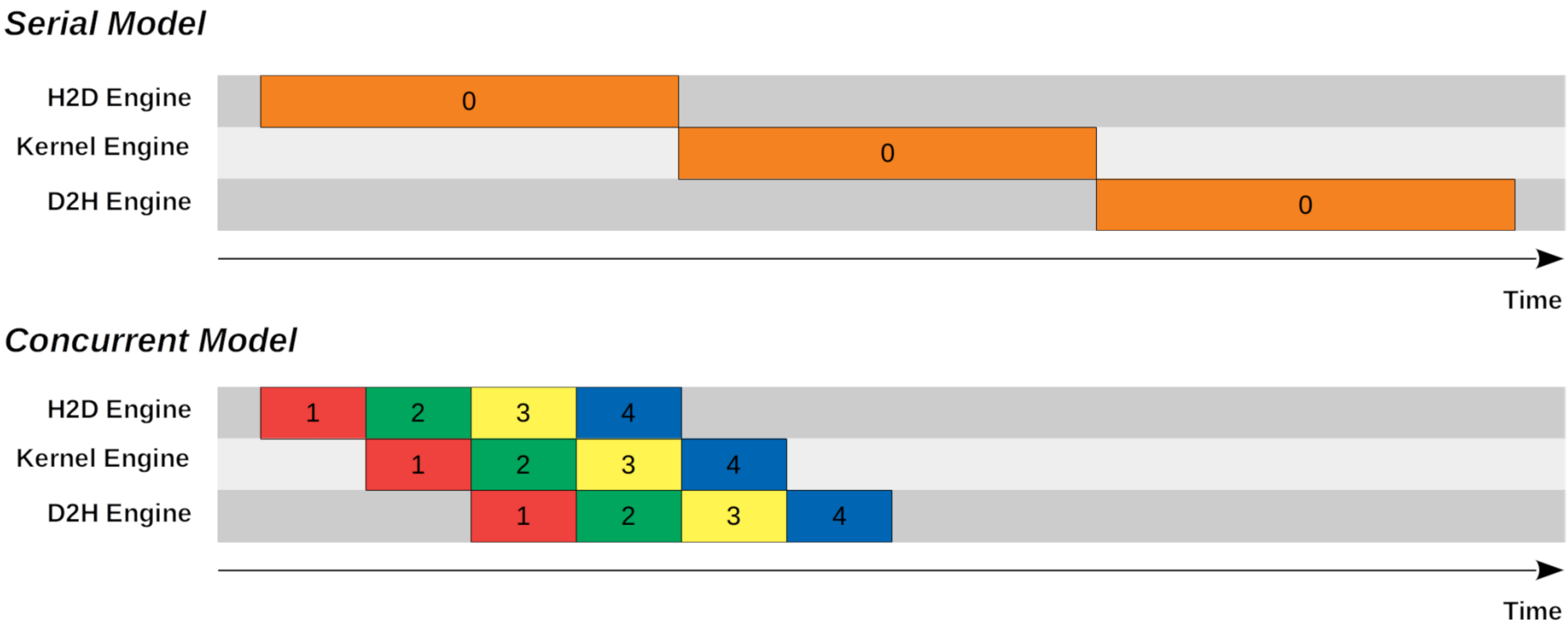
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
cudaMemcpyAsync(d_a, h_a, N, cudaMemcpyHostToDevice, stream1);


kernel<<<1,N,0,stream1>>>(d_a)

|
cudaMemcpyAsync(h_a, d_a, N, cudaMemcpyDeviceToHost, stream1);
cudaStreamDestroy(stream1);
```

Streams (Sincronização)

A função **cudaStreamSynchronize(stream)** pode ser usada para bloquear o host até que todas as operações emitidas anteriormente no stream especificado sejam concluídas.
A função **cudaStreamQuery(stream)** testa se todas as operações emitidas para o stream especificado foram concluídas, sem bloquear a execução do host.

Streams

Intercalando envio de dados e execução kernel



Último Slide

- Obrigado!
- Dúvidas?
- Email: pedro.lara@cefet-rj.br