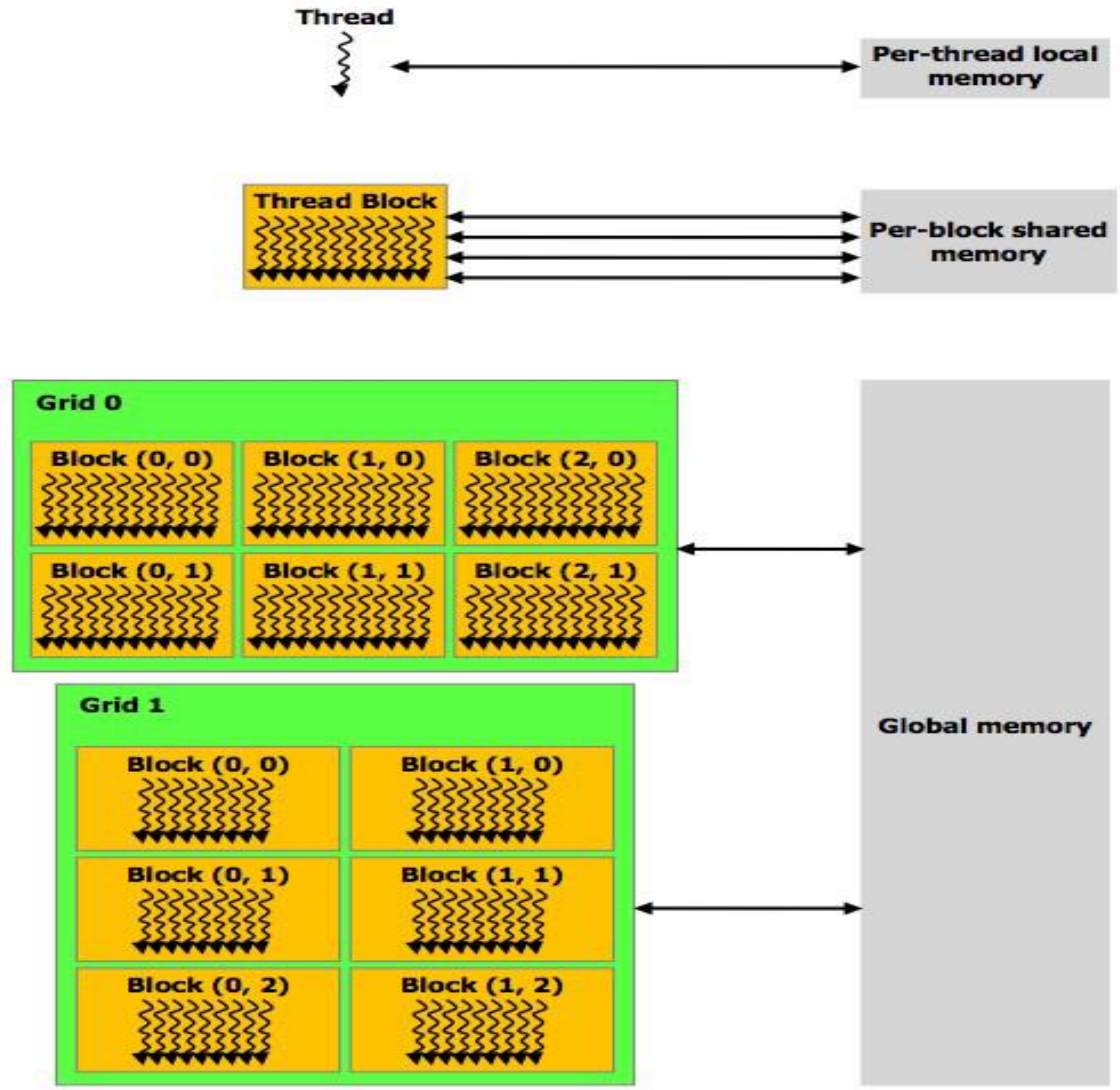
INE5645 – Programação Paralela e Distribuída
Aluno _____

Modelo de Execução CUDA - A execução do programa controlado pela CPU pode lançar kernels, que são trechos de código executados em paralelo por múltiplas threads na GPU.

A execução de programas CUDA é composta por ciclos:
CPU, GPU, CPU, GPU, … , CPU, GPU, CPU.

Hierarquia de memória:



Cada execução do kernel é composta, numa visão geral:

Grid → blocos → threads

Hierarquia de memória:

- Registradores por thread.
- Memória compartilhada por bloco.
- Memória Global acessível a todas as threads.

Exemplo Simples

Neste primeiro exemplo iremos aprender como criar um _____(kernel) simples, que realiza a soma de 2 vetores.

Veremos as principais operações usadas em CUDA:

(1) Alocação de _____ (memória).
(3) Transferência de _____ (dados).
(4) Lançamento do _____ (kernel).
(2) Liberação de _____ (memória).

O código abaixo contém erros e limitações.

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

A primeira coisa a notar é a palavra-chave **__global__** significa a construção de um _____ em CUDA. Isso simplesmente indica que essa função pode ser chamada do Host ou do dispositivo CUDA.

A próxima coisa que você deve notar é como cada thread (_____) descobre exatamente qual elemento de dados é responsável pela computação.

Cada thread executa o mesmo código, portanto, a única maneira de se diferenciar threads é usar o **threadIdx** (_____) e o **blockIdx** (_____).

O valor de **threadIdx.x** e **blockIdx.x** devolve o _____ e o bloco ao qual a thread pertence.

```
// Host code
// "h significa host, enquanto d significa device)"

int main() {
 int n = 5;
 size_t size = n * sizeof(float);
 float *d_A, *d_B, *d_C;
```

// **"void*"** é um ponteiro para algo. Mas **cudaMalloc()** precisa modificar o ponteiro dado (o próprio ponteiro, não para o qual o ponteiro aponta), então você precisa passar **"void **"** que é um ponteiro para o ponteiro (geralmente um ponteiro para a variável local que aponta para o endereço de memória) tal que **cudaMalloc()** pode modificar o valor do ponteiro.

```
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);
cudaMalloc((void**)&d_C, size);
```

// Entrada de dados dos vetores A e B via Host.

```
float h_A[] = {1,2,3,4,5};
float h_B[] = {10,20,30,40,50};
float h_C[] = {0,0,0,0,0};
```

// Copia os vetores A e B do Host para o Device.

```
cudaMemcpy(d_A, h_A, size,
     cudaMemcpyHostToDevice);

cudaMemcpy(d_B, h_B, size,
     cudaMemcpyHostToDevice);
```

// Define o número de threads por bloco.

```
int nThreadsPerBlock = 256 (múltiplo de 32);
int nBlocks = n/nThreadsPerBlock; ????
```

// Chamada do kernel.

```
 VecAdd<<<nBlocks,
   nThreadsPerBlock>>>(d_A, d_B, d_C);
```

// Copia o vetor C da memória da GPU para a memória no HOST.

```
 cudaMemcpy(h_C, d_C, size,
            cudaMemcpyDeviceToHost);
```

```
// Libera memória ocupada pelos vetores.
 cudaFree(d_A);
 cudaFree(d_B);
 cudaFree(d_C);
}
```

Organizando Threads

Uma parte crítica do projeto de aplicativos CUDA é organizar threads, blocos de threads e grids de forma apropriada.

Para este aplicativo, a escolha mais simples é fazer com que cada thread calcule um elemento (uma entrada), e apenas uma thread, no array do resultado final.

Uma orientação geral é que um bloco deve consistir em pelo menos 192 threads para ocultar a latência do acesso à memória (Tempo de latência, é o tempo que ela demora para entregar os dados... são os tempos de espera para troca de dados... sendo assim, quanto menor o tempo para a entrega, mais rápido fica**).**

Portanto, 256 e 512 threads são números comuns e práticos. Para os propósitos deste exemplo, são selecionadas 256 threads por bloco.

Se nós temos n elementos de dados, nós necessitamos somente n threads, no sentido de computar a soma dos vetores ...

Assim, necessitamos o menor múltiplo de threadsPerBlock que é maior ou igual a n, por computar:

(n + (threadsPerBlock-1))/threadsPerBlock

Assim, o número de blocos lançados nBlocks deve ser 32 ou (n + (threadsPerBlock - 1))/threadsPerBlock .

https://books.google.com.br/books?id=.....

Para compilar o código, basta utilizar o comando:

```
nvcc -o ex1 ex1.cu
```

Perguntas:

1) O programa funciona corretamente?
   Teste seu funcionamento imprimindo o resultado obtido.

2) Corrija o programa.

3) Aumente o tamanho dos vetores para, por exemplo, 1024. Teste o resultado e, se não for o esperado, corrija o programa. Obs: Você deve manter o número de threads por bloco em 256.

Referências:

https://eradsp2010.files.wordpress.com/2010/10/curso2_cuda_camargo.pdf

https://en.wikipedia.org/wiki/CUDA

http://supercomputingblog.com/cuda/cuda-tutorial-2-the-kernel/