Introduction to the CUDA Programming Language

Compute Unified Device Architecture Execution Model Overview

- Architecture and programming model, introduced in NVIDIA in 2007.

- Enables GPUs to execute programs written in C in an integrated **host (CPU) + device (GPU)** app C program.
  - Serial or modestly parallel parts in **host** C code.
  - Highly parallel parts in **device** SIMT codes kernel code.

- Differences between GPU and CPU threads:
  - GPU threads are extremely lightweight with very little creation overhead.
  - GPU needs 1000's of threads for full efficiency (multi-core CPU needs only a few).

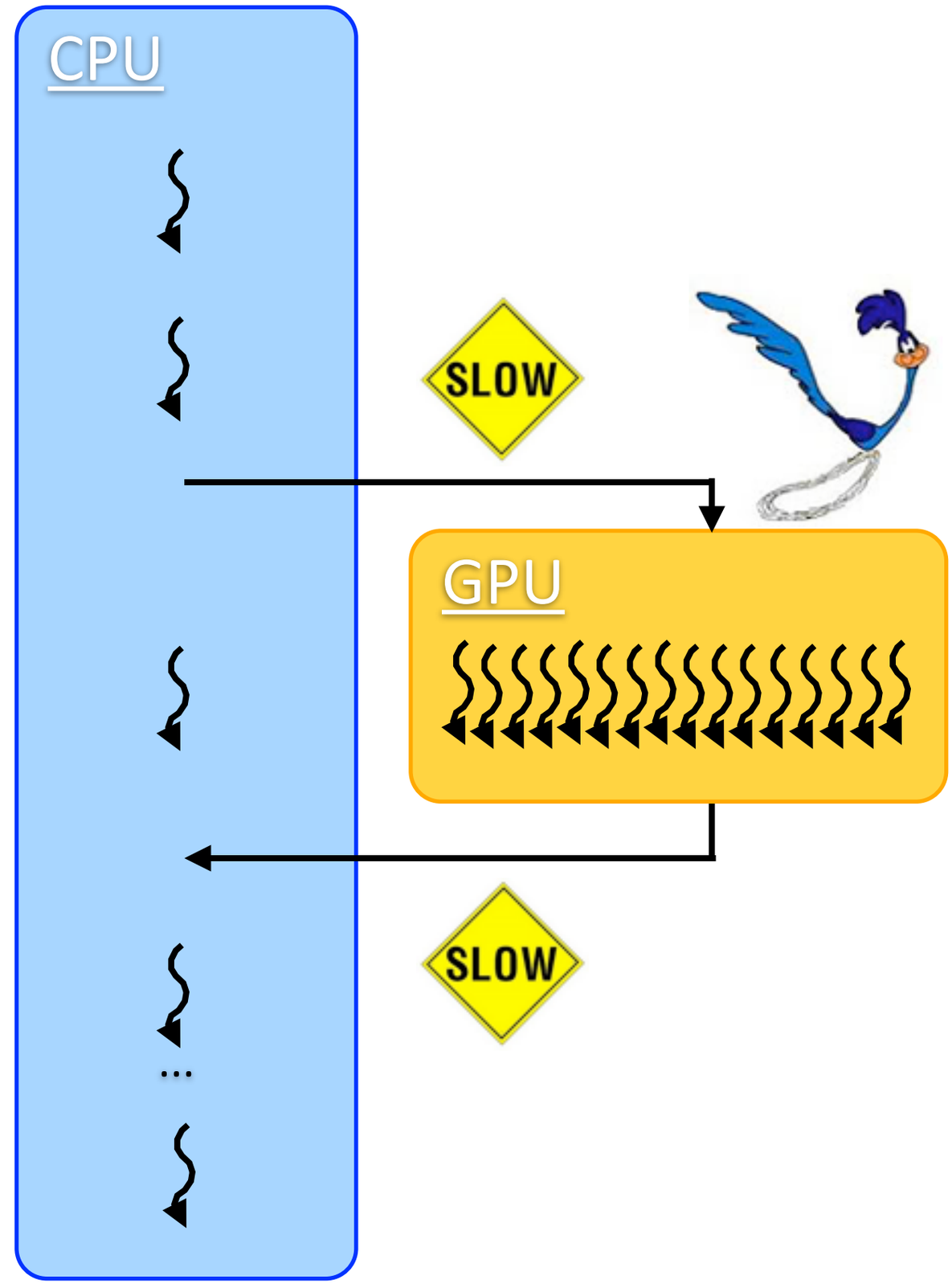
Compute Unified Device Architecture
Execution Model Overview

C/CUDA Code

```
// serial code
int main() {
 printf("Hello world!"\n);
// allocate data
 cudaMalloc(...);
// copy data
 cudaMemcpy(...);
// execute kernel

 cudaRun<<<..>>>(...);

  ...

 cudaThreadSynchronize();

// serial code
 printf("Running.."\n);

  ...

 exit (0);
}
```

CUDA Kernel Code

```
// kernel
___global___
void cudaRun(...) {
  ...
}
```



Hello World v.1.0: Basic C Program

```
#include <stdio.h>

int main() {
 printf("Hello world!"\n);
 exit (0);
}
```

```
nvcc -o hello hello.cu
```

OUTPUT:

```
Hello World!
```

Compiling CUDA Code

directories for the #include files

- **nvcc –o <exe> <source_file> -I/usr/local/cuda/include**

  –L/usr/local/cuda/lib –lcuda –lcudart

  directories for libraries

  libraries to be linked

- If a Cuda code includes device code, the file must have the extension .cu.
- nvcc separates out code for the CPU and code for the GPU and compiles code.
  - It needs regular C compiler installed for the CPU code.

Executing a CUDA Program

- **./a.out**

- Host code starts running.

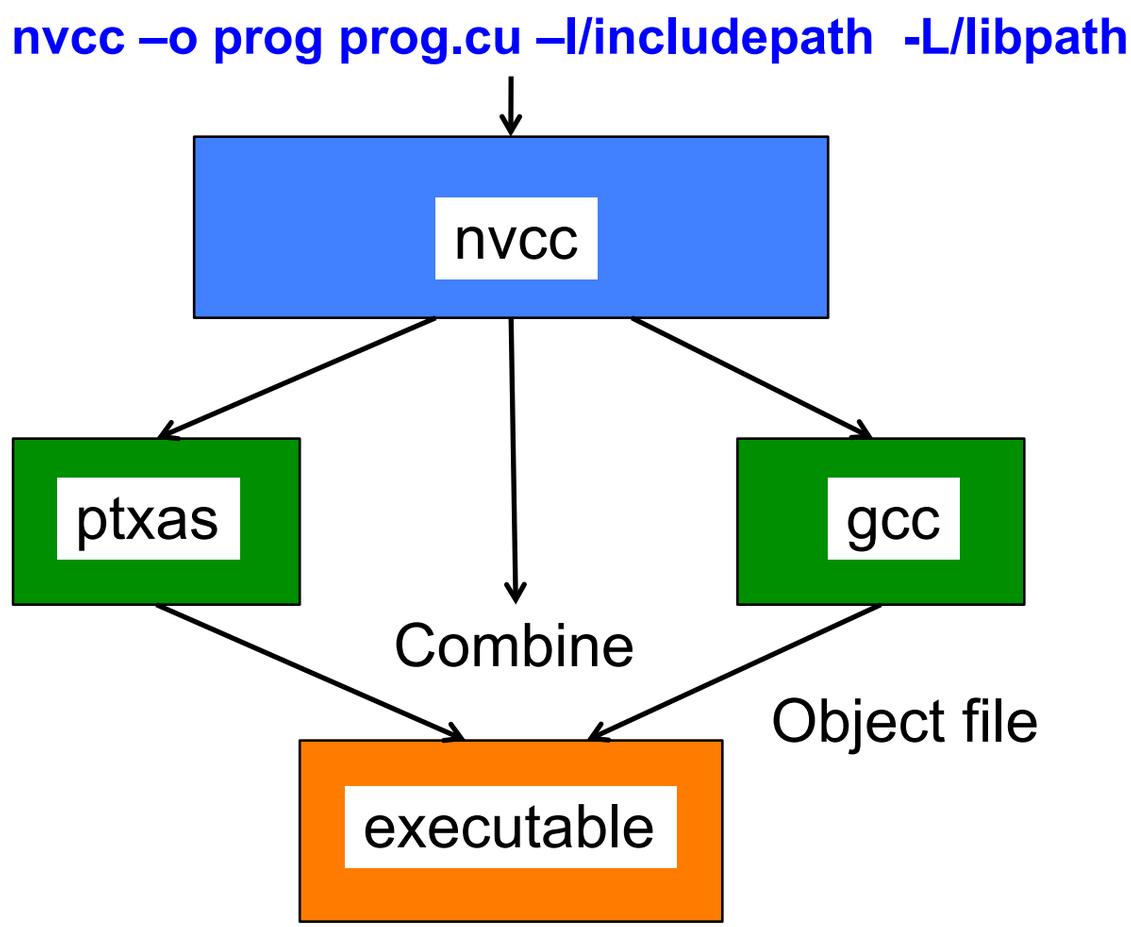
- When first encounter device kernel, GPU code physically sent to GPU and function launched on GPU.

- Hence first launch will be slow!!

Compilation Process

ptxas = PTX assembler = parallel thread execution

- nvcc "wrapper" divides code into host and device parts.
- Host part compiled by regular C compiler.
- Device part compiled by the NVIDIA "ptxas" assembler.
- Two compiled parts combined into one executable.



Executable file a "fat" binary" with both host and device code

Hello World v.2.0: Kernel Calls

```
#include <stdio.h>

int main() {
 kernel<<<1,1>>>();
 printf("Hello world!\n");
 exit (0);
}

__global__ void kernel () {
  // does nothing
}
```

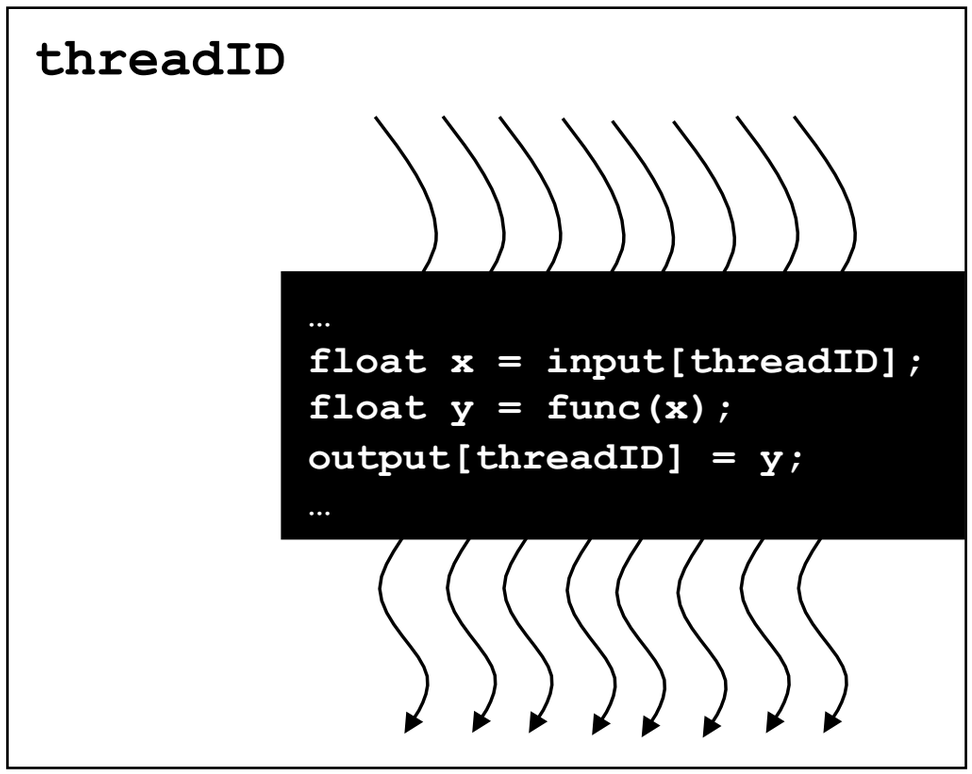
- An empty function named "kernel" qualified with the specifier __global__ (yes, there are two underscores on each side)
  - Indicates to the compiler that the code should be run on the device, not the host.
- A call to the empty device function with "<<<1,1>>>"
- Within the "<<<" and ">>>" brackets are **memory arguments** (for the blocks and threads) and within the parentheses are the **parameter arguments** (that you normally use in C).

```
OUTPUT:

Hello World!
```

CUDA Kernel Routines

- A kernel routine is executed on the device. This one set of instructions is executed by each allocated thread (SIMT).

- The kernel code is mostly the same as C. One exception is that each kernel code has an associated thread ID so that it can access different data (more on this later).



CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

Hello World v.3.0: Parameter Passing

```
#include <stdio.h>

__global__ void add (int a, int b, int *c);

int main() {
 int c;
 int *dev_c;

 cudaMalloc((void**)&dev_c, sizeof(int));

 add<<<1,1>>>(2,7,dev_c);

 cudaMemcpy(&c, dev_c, sizeof(int),
            cudaMemcpyDeviceToHost);

 printf("Hello world!\n");
 printf("2 + 7 = %d\n", c);

 cudaFree(dev_c);

 exit (0);
}

__global__ void add (int a, int b, int *c) {
 c[0] = a + b;
}
```

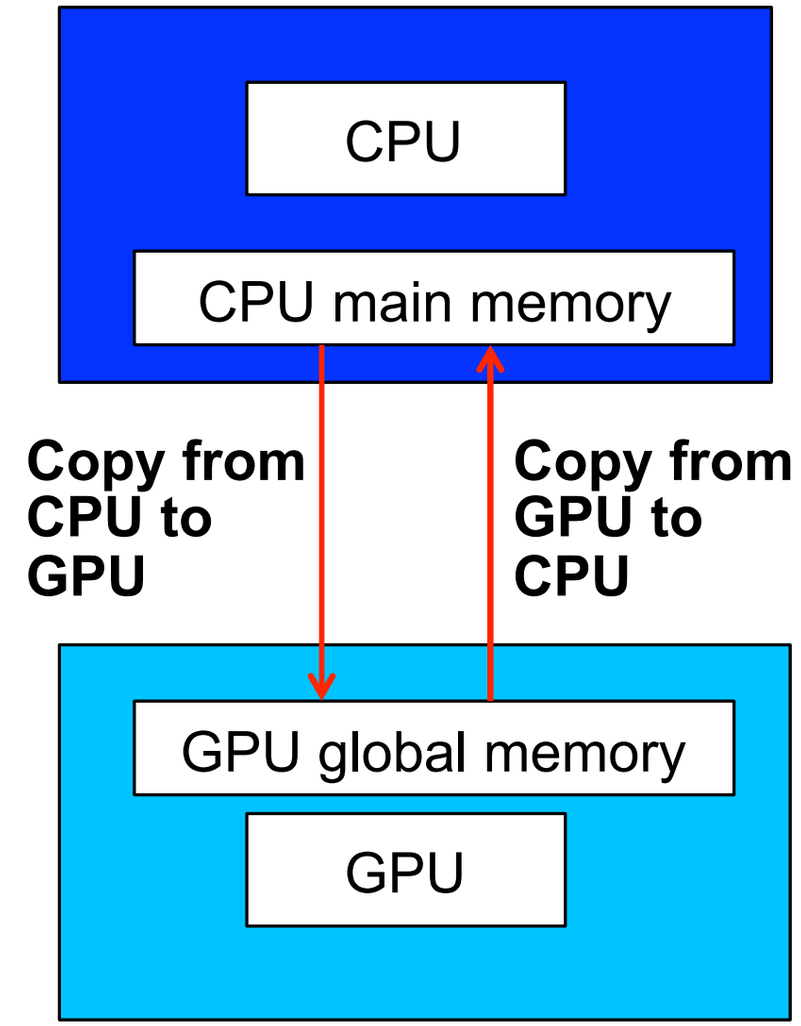
- Parameter passing is similar to C.
- There exists a separate set of host + device memory.
- We need to allocate memory to use it on the device.
- We need to copy memory from the host to the device and/or vice versa via cudaMemcpy.
- CudaMalloc (similar to malloc) allocates global memory on the *device*.
- CudaFree (similar to free) deallocates global memory on the *device*.
- Of course, capable of mathematic operations.

```
OUTPUT:

Hello World!
2 + 7 = 9
```

CPU and GPU Memory

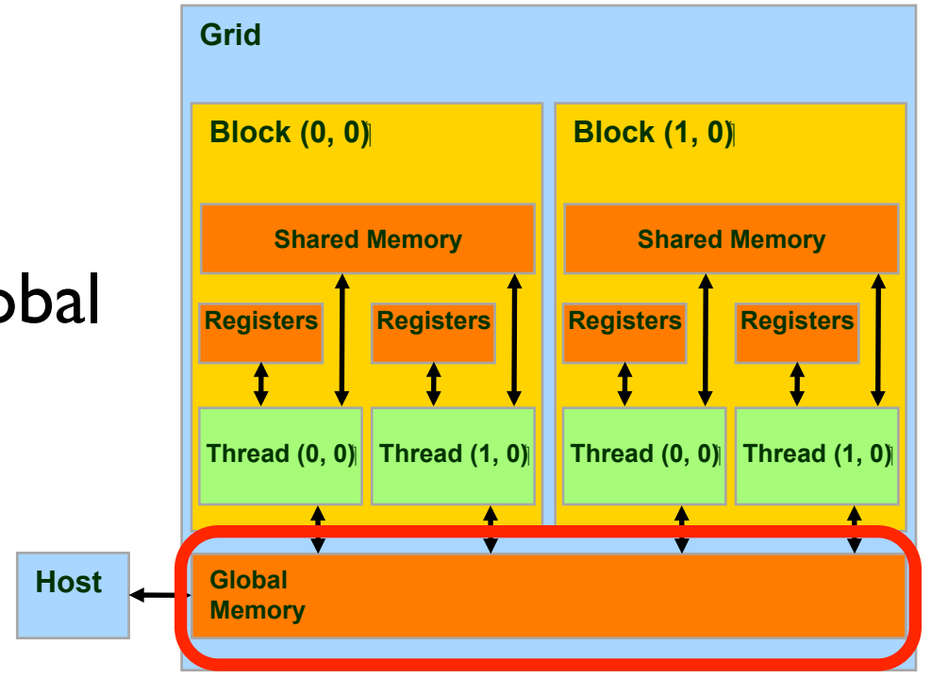
- A compiled CUDA program has:
  - 1) code executed on the CPU
  - 2) (kernel) code executed on the GPU

- The CPU and GPU memories are distinct and separate. Therefore, we need to:
  - 1) allocate a set of host data and device data and
  - 2) explicitly transfer data from the CPU to the GPU and vice versa.



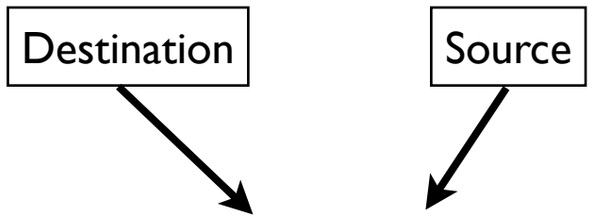
Allocating and Deallocating Device Memory Space via cudaMalloc and cudaFree

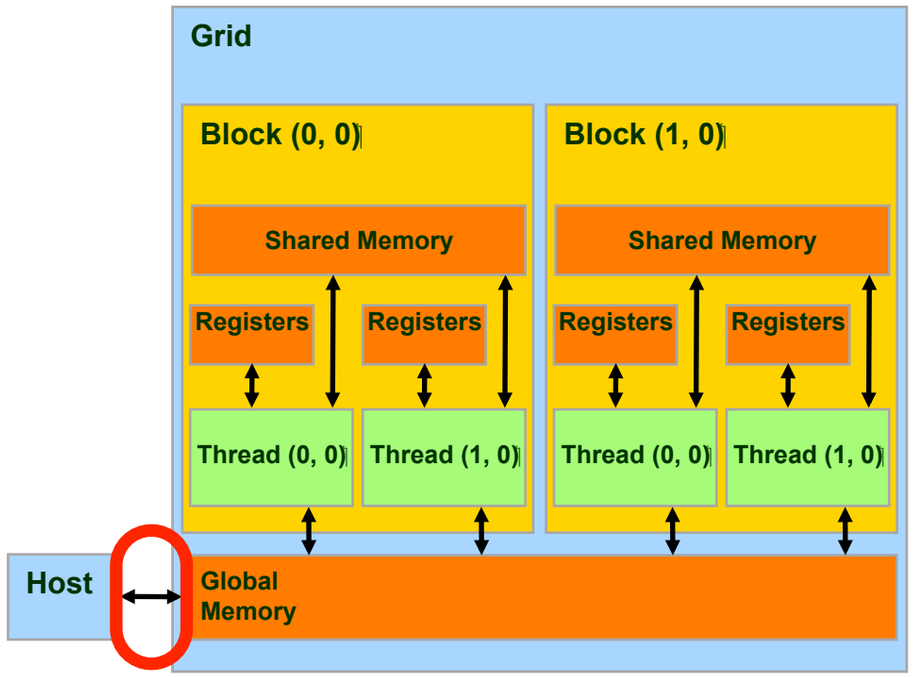
- Allocating Memory: allocates object in device global memory and returns pointer to it.
  - `int *dev_C;`
  - `int size = N *sizeof( int);`
  - `cudaMalloc((void**)&dev_C, size);`



- Deallocating Memory: free object from device global memory
  - `cudaFree(dev_C)`

Transferring Data via cudaMemcpy





- `cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);`

- `cudaMemcpy( c, dev_c, size, cudaMemcpyHostToDevice);`

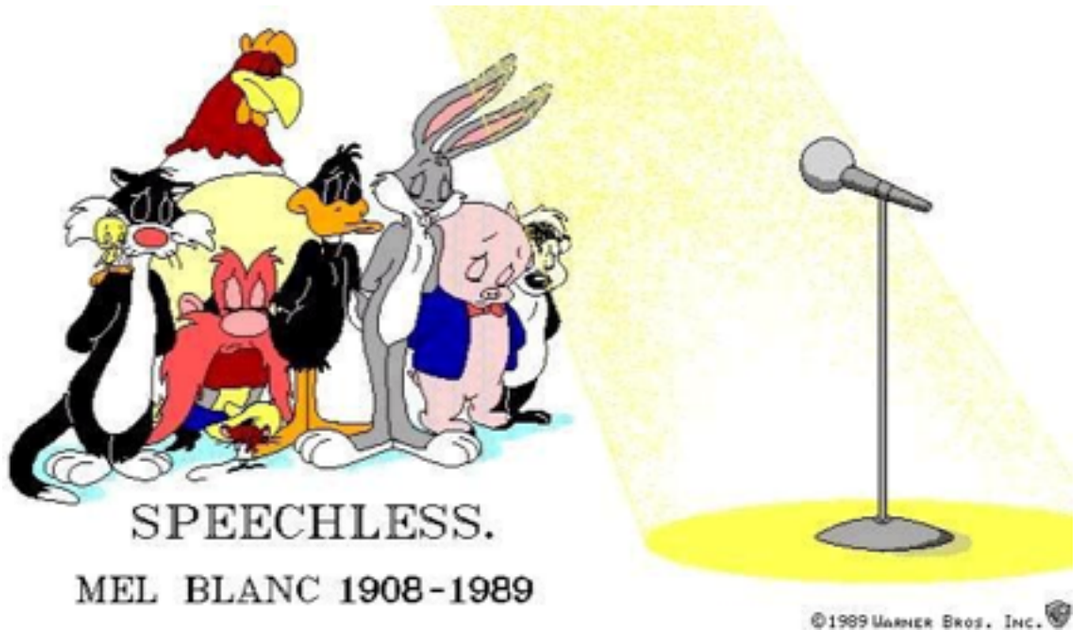
- "dev_a" and "dev_c" are pointers to device data
- "a" and "c" are pointers to host data
- "size" is the size of the data
- "cudaMemcpyHostToDevice" and "cudaMemcpyDeviceToHost" tells cudaMemcpy the source and estination of the operation.

Hello World v.4.0: Vector Addition

```
#define N 256
#include <stdio.h>

__global__ void vecAdd (int *a, int *b, int *c);

int main() {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;

 // initialize a and b with real values (NOT SHOWN)

 size = N * sizeof(int);

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);

 cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

 vecAdd<<<1,N>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);

 exit (0);
}

__global__ void vecAdd (int *a, int *b, int *c) {
 int i = threadIdx.x;
 c[i] = a[i] + b[i];
}
```

- CUDA gives each thread a unique ThreadID to distinguish between each other even though the kernel instructions are the same.
- In our example, in the kernel call the memory arguments specify 1 block and N threads.

```
OUTPUT:
```



GPU Thread Structure

- A CUDA kernel is executed on an **array or matrix** of threads.
  - All threads run the same code (SIMT)
  - Each threads has an ID that is uses to compute memory addresses and make control decisions.
    - Block ID: 1D or 2D
    - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data