LESSON 1

First kernel

TOPICS

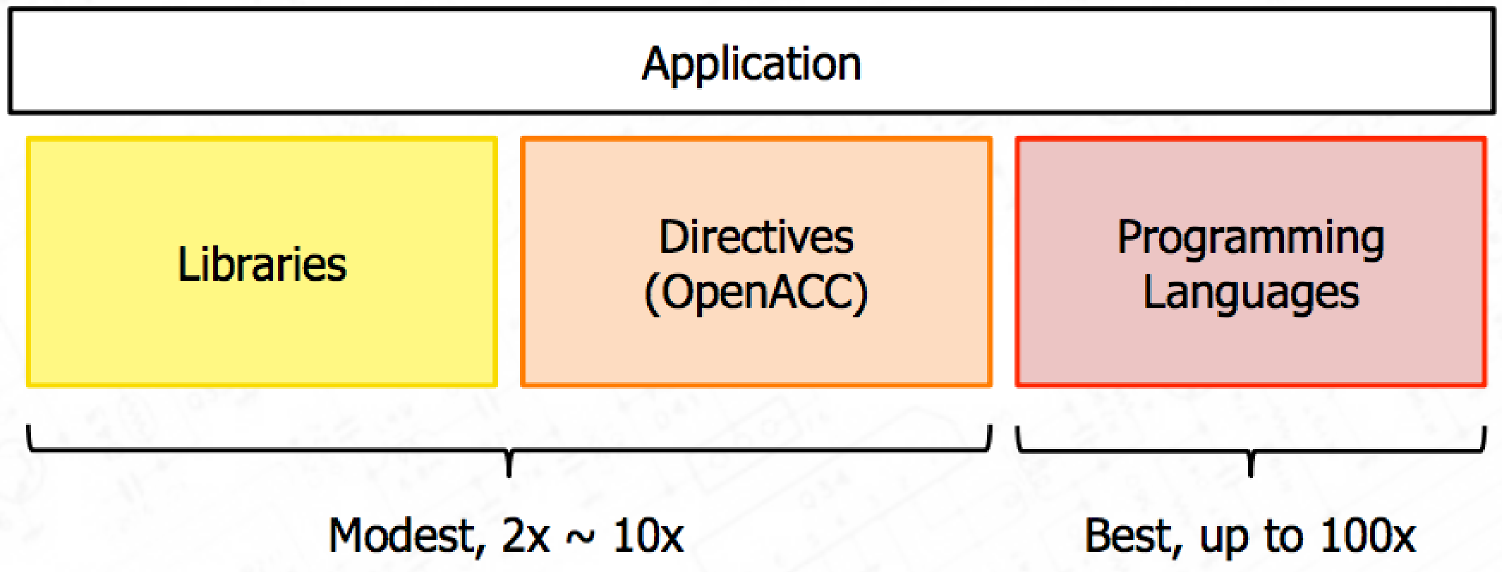
- How to identify the basic terminology used in CUDA (Compute Unified Device Architecture)
- Difference between device (GPU) and host (CPU)
- Code your first application to be executed in the device

Key words: cudaMalloc, cudaFree, threaIdx, cudaMemcpy, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, nvcc.

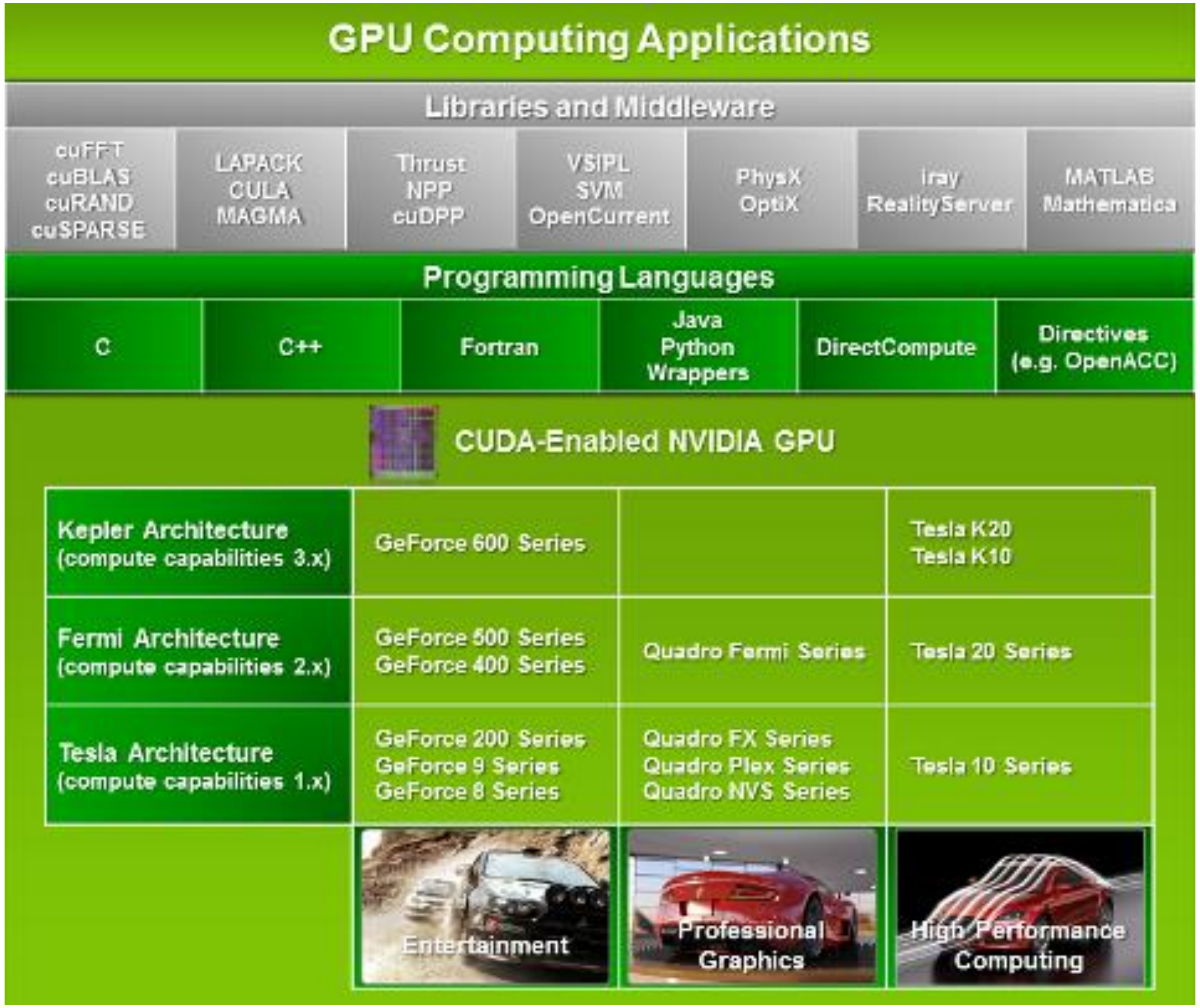
CUDA

- GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about.
- It is a parallel programming paradigm, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations.
- CUDA, a General-Purpose Parallel Computing Platform and Programming Model
- CUDA comes with a software environment that allows developers to use C as a high-level programming language.

- CUDA runtime system provides an application-programming interface (API) to start working
- API are functions that help users to perform some of the functionalities.
- http://docs.nvidia.com/cuda/



- CUDA ecosystem



Kernel

- CUDA C extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C functions.

- This kind of function indicates to the compiler that the code will run on the device, and it's callable from the host.

```
__global__ void function(parameter0, parameter1,…,
parameterN){

...

}
```

- A call to a __global__ function is asynchronous, meaning it returns before the device has completed its execution.

- To invoke the kernel function, it is required to use angle brackets <<<n,n>>>.

```
function<<<1,1>>>(parameter1, parmeter2,…, parameterN)
```

- The angle brackets indicate how the runtime will launch the device code.

Compilation

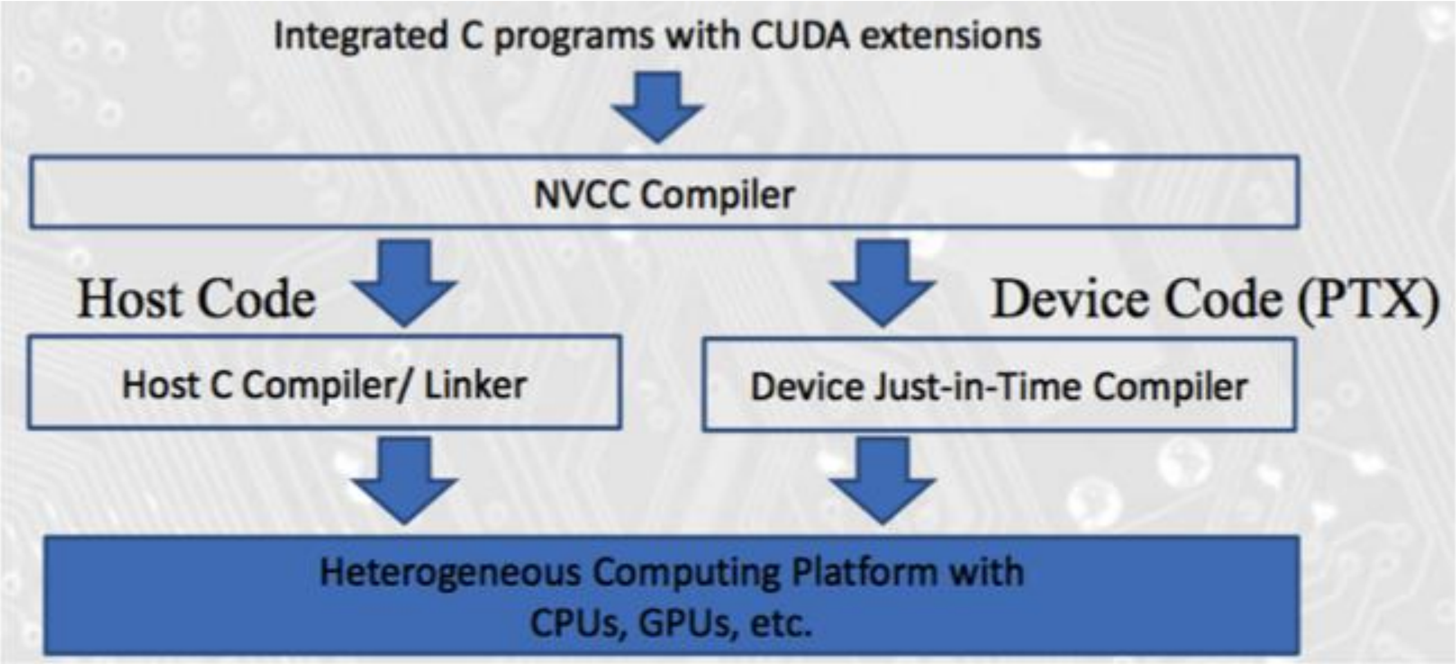
- To compile the example we need to call the Nvidia compiler:

```
$ nvcc  source_code.cu   -o  exe
```

- To run the example:

```
$ ./exe
```

- Kernels can be written using the CUDA instruction set architecture, called PTX.
- nvcc is a compiler driver that simplifies the process of compiling C or PTX code: It provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages.



Compilation Workflow

- Source files compiled with nvcc can include a mix of host code (i.e., code that executes on the host) and device code (i.e., code that executes on the device).

- nvcc's basic workflow consists in separating device code from host code.
  - compiling the device code into an assembly form (*PTX* code) and/or binary form (*cubin* object)
  - and modifying the host code by replacing the <<<...>>> syntax by the necessary CUDA C runtime function calls to load and launch each compiled kernel from the *PTX* code and/or *cubin* object.

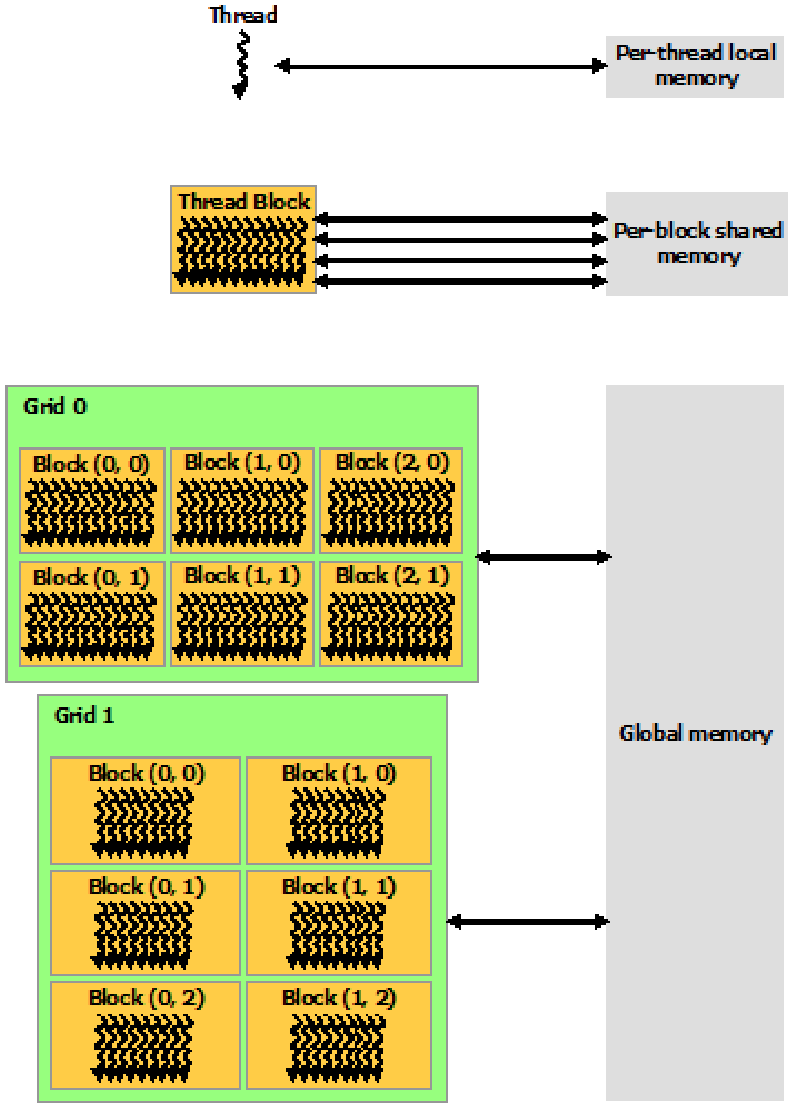
- The modified host code is output either as C code that is left to be compiled using another tool or as object code directly by letting nvcc invoke the host compiler during the last compilation stage.

Example

- 01simple_kernel: This example only creates a kernel to be called, but it doesn't execute any actual instruction in the device.

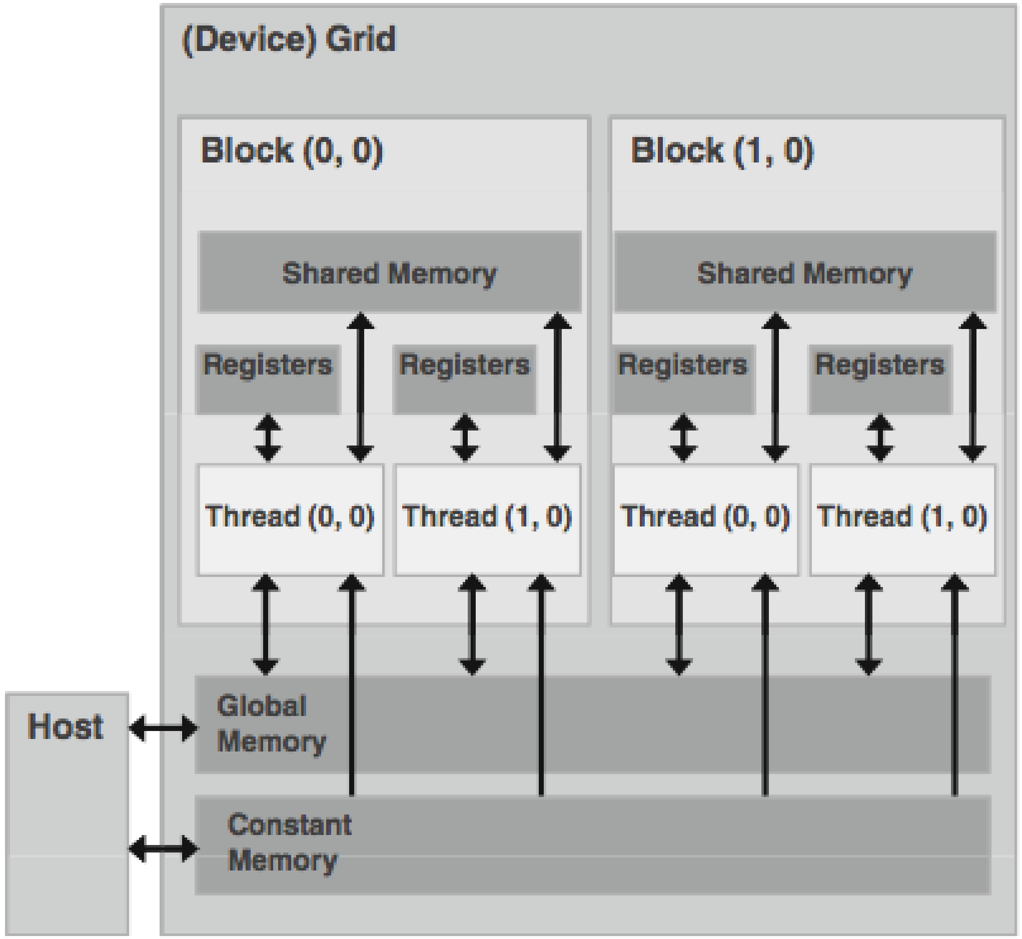
Memory Hierarchy

- Each thread has private local memory.
- Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.
- All threads have access to the same global memory.



CUDA device memory

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories



cudaMalloc

- Besides the kernel function, it is necessary to allocate memory on the device for the parameters that the kernel needs.

```
cudaError_t   cudaMalloc(void** devPtr, size_t size)
where:
devPtr - Pointer to allocated device memory
size – Requested allocation size in bytes
```

- Host pointers can access memory from the host code, and device pointers can access memory from the device code.

- This memory, allocated with cudaMalloc() is called linear memory.

cudaFree

- Similar to C programming language, it is necessary to free the memory that is no longer needed, but it must be the one allocated on the device.

```
cudaError_t cudaFree (void * devPtr)
```

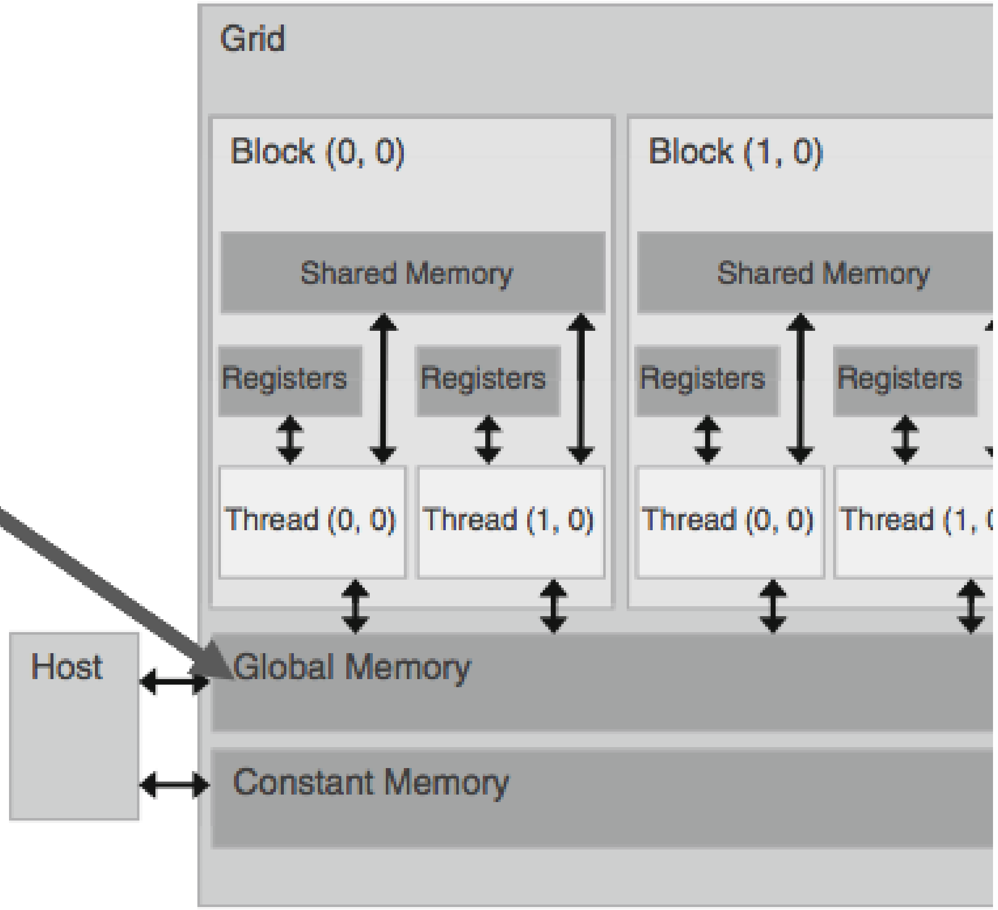
CUDA device memory

cudaMalloc()

- Allocates object in the device global memory
- Two parameters
  - **Address of a pointer** to the allocated object
  - **Size of** of allocated object in terms of bytes

cudaFree()

- Frees object from device global memory
  - Pointer to freed object



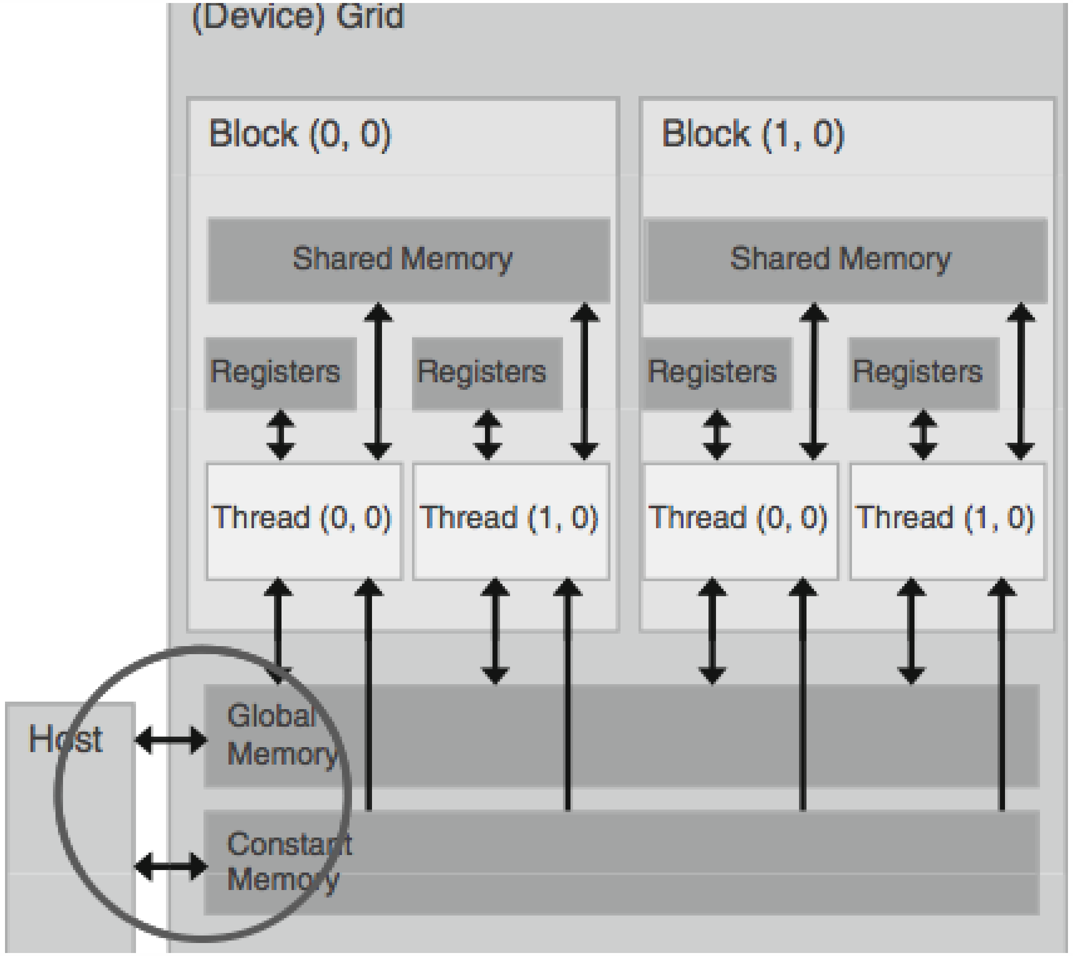
cudaMemcpy

- Finally to copy the memory on the device, the *cudaMemcpy()* function is employed.

```
cudaError_t  cudaMemcpy( void * dst, const void * src, size_t
count, enumcudaMemcpyKind kind)
where:
dst - Destination memory address
src – Source memory address
count – Size in bytes to copy
kind – Type of transfer (cudamemcpyHostToDevice,
cudamemcpyDeviceToHost,)
```

CUDA device memory

- cudaMemcpy()
  - **Memory** data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Transfer is asynchronous



Example

- 02simple_kernel2: This code sends three parameters to the kernel and executes the addition operation on them.