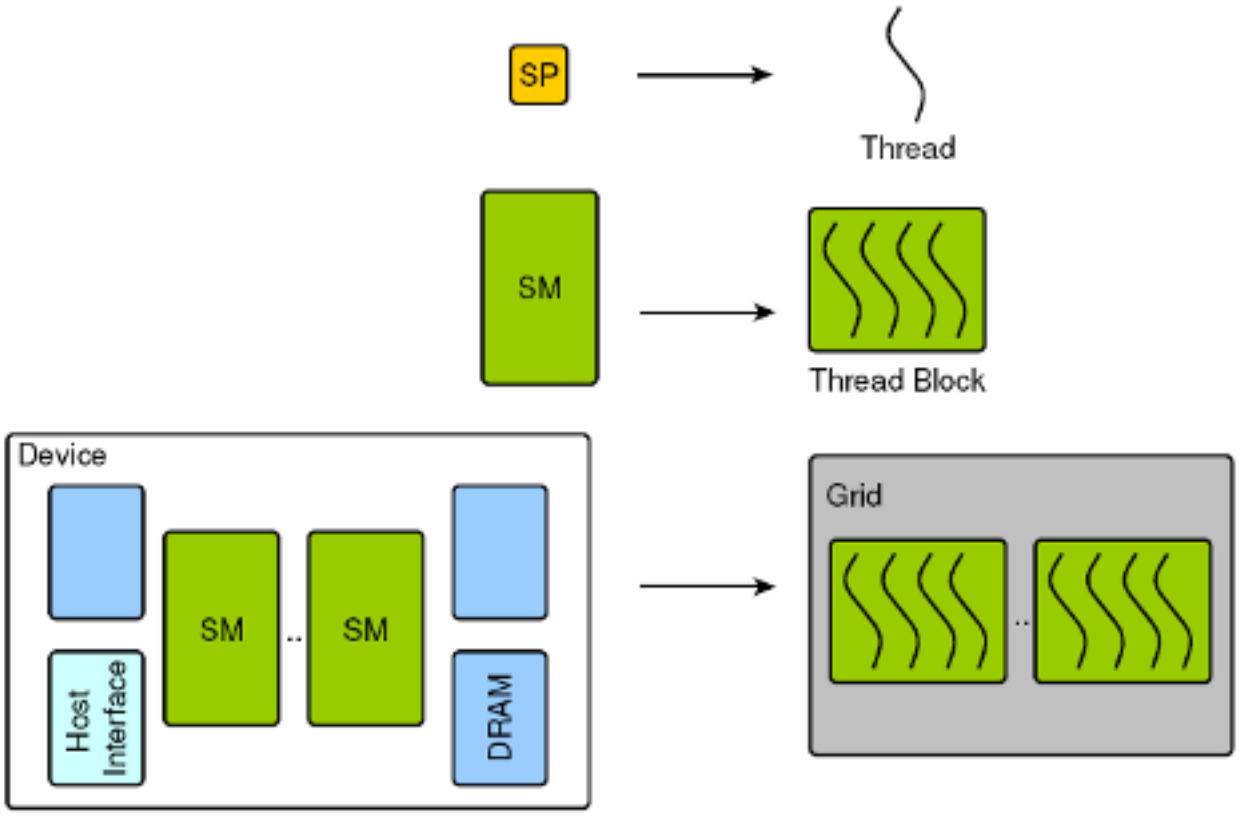
GPU Programming - CUDA

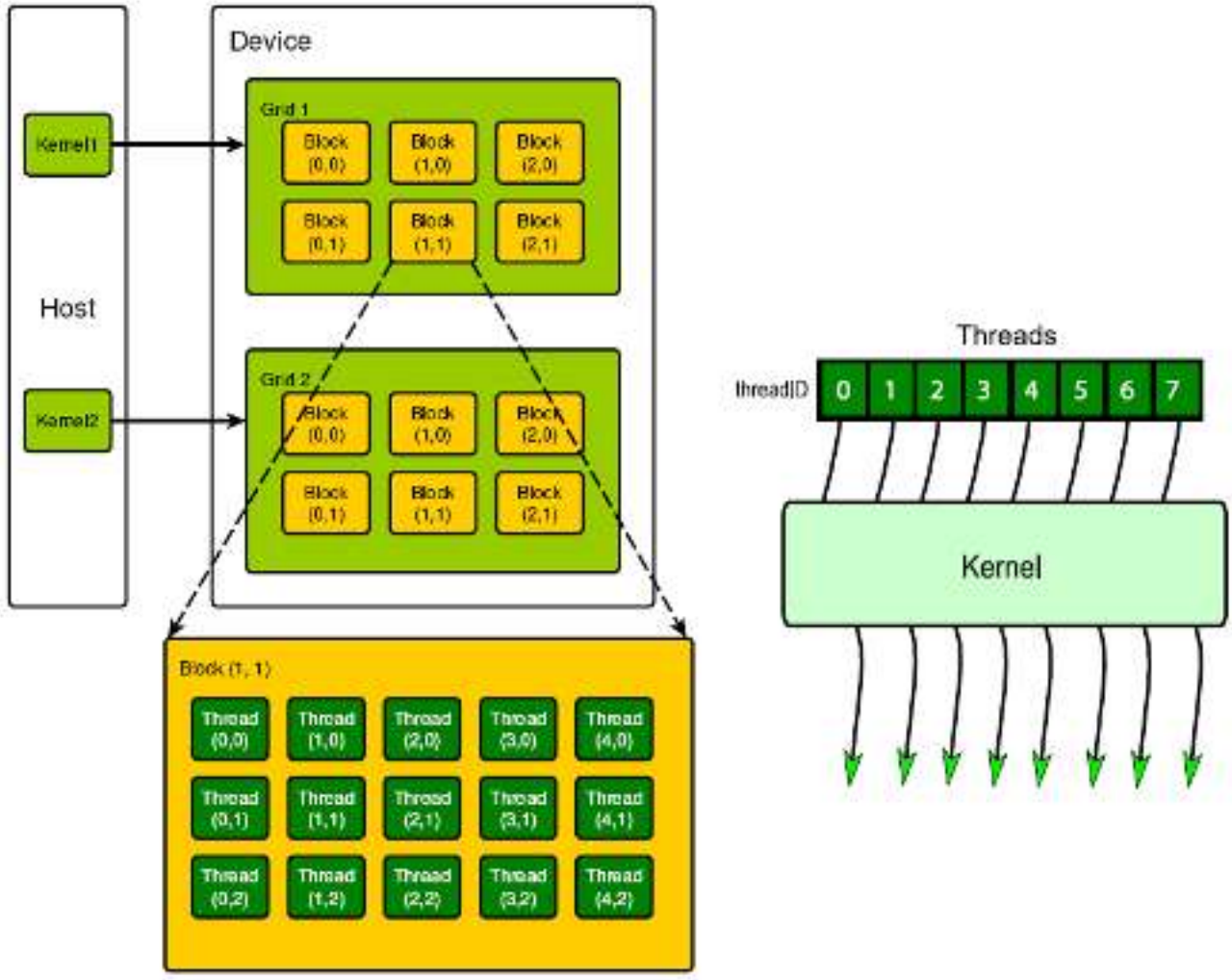
CUDA provides a relatively simple C-like interface to develop GPU-based applications. It exposes an abstraction to the programmers that completely hides the underlying hardware architecture. As a result, they see any CUDA-enabled GPUs as a collection of a number of threads organised into blocks and a collection of blocks that are organised into a grid. At the hardware level, the threads and blocks are handled by the streaming processors (SP) and streaming multiprocessors (SM), respectively, and the device is mapped as a **grid**. A simplified schematic diagram of the mapping is illustrated in the figure below:



Mapping of CUDA threads, blocks and grid

A CUDA program typically consists of a component that runs on the CPU (host), and a smaller, but computationally intensive component called kernel that runs in parallel on the GPUs (device). Programmers can define how many

parallel threads will execute the kernel, however, there is a limit to the maximum number of threads per block (since all threads of a block must reside inside the same SM). Currently, Tesla GPUs allow up to 1024 threads per block. Blocks can have 1,2, or 3-dimensional orientation of threads and they can be organised into a 1,2-dimensional grid (a 2D orientation of threads and blocks is illustrated in Figure (a)). Even though all the threads and blocks are designed to execute exactly one kernel function at a time (Figure (b)), they can be assigned to a specific portion of the data to work on. CUDA provides built-in variables that facilitate the identification of each of the blocks and threads separately during the kernel execution (Figure ©).




```
int bx = blockIdx.x, by = blockIdx.y;
int tx = threadIdx.x, ty = threadIdx.y;
```

(c)

Figure (a) An illustrative thread and block organization (b) SIMT based kernel execution (c) Built-in variables to identify CUDA threads and blocks.

In terms of memory access, a kernel cannot access the main memory of the host directly (i.e., we need a host CPU to do the memory management). The input data for the kernel must be copied to the GPU's on-board memory prior to its invocation and the output data from a kernel must first be written to the GPU's memory, and then copied back to the host CPU memory (Figure (a)). CUDA supports different types of memory and each of them has specific scope (Figure (b)) and life span, such as registers (allocated per thread, handled by the 327,68 32-bit registers in each SM), constant (allocated to all the threads, embedded into the executable code, does not consume any register), local (allocated per thread, physically handled by the DRAM), shared (allocated per thread block, handled by L1 cache by default) and global (allocated to all the threads and blocks, handled by the DRAM). In addition, there is a read-only texture memory, which is purely intended to handle graphical applications.

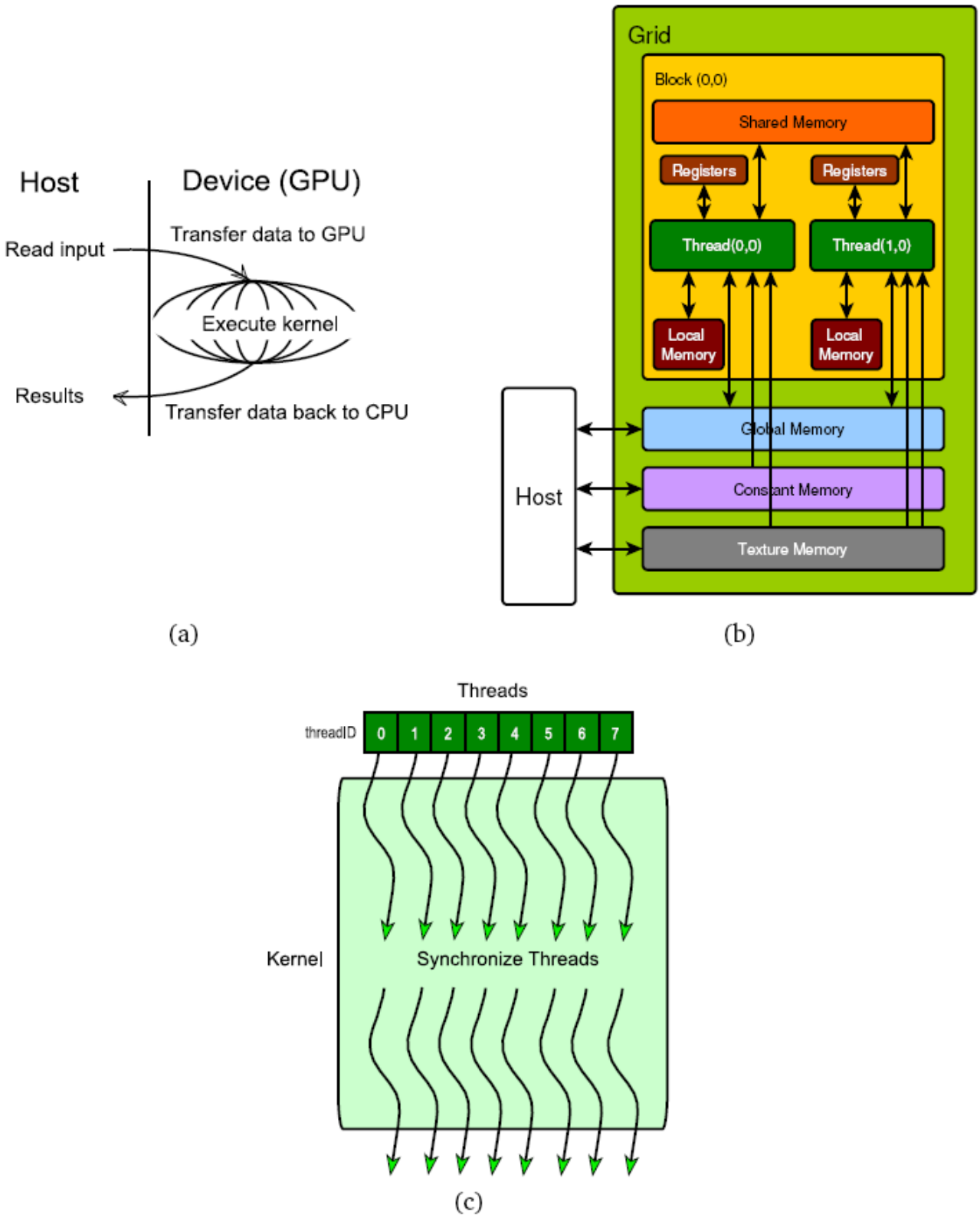


Figure (a) CUDA data allocation for a kernel (b) Memory scopes (c) Thread synchronization

Recent developments of the CUDA API also allow critical sections, which are implemented as atomic operations. These operations can assist a programmer to avoid any race conditions that may arise when two or more threads try to write in a single memory location. They are capable of read-modify-and-write a value back to the device memory without any interference from other parallel threads, and which guarantees that the race condition will not occur.