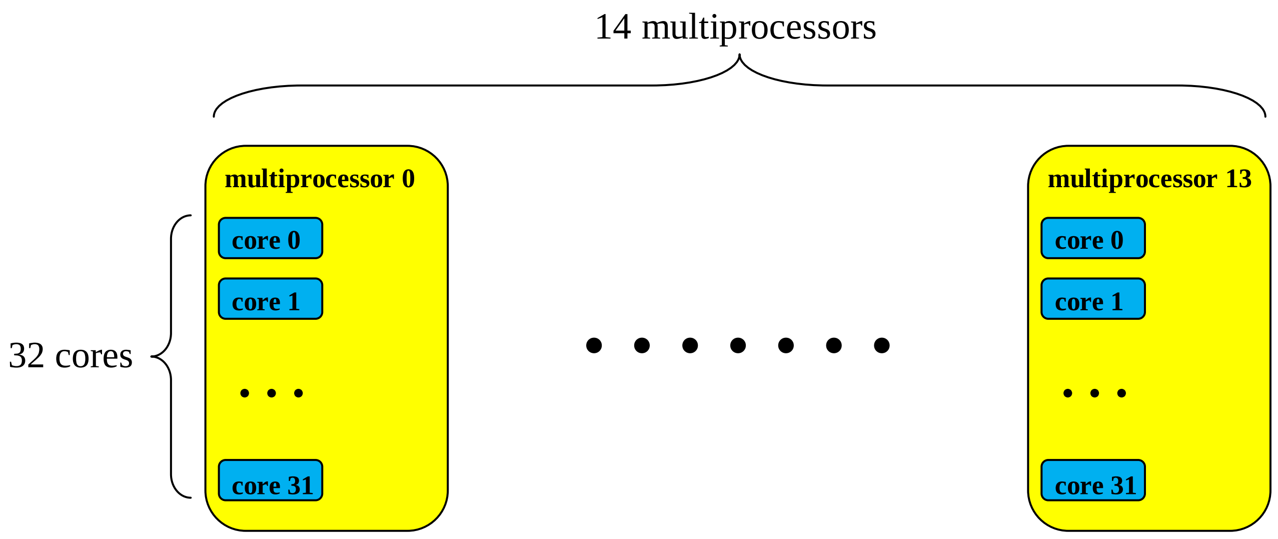
CUDA architecture and programming model

CUDA architecture (compute capability 2.0)

shared memory system:

- *two processing levels:*



= 14 × 32 = **448 cores in all**

- *two corresponding levels of shared memory:*
  - one *global memory* (3 or 6 GB) = shared by all the cores of the 14 multiprocessors
  - one *shared memory* (16 or 48 kB) for each multiprocessor = shared by its 32 cores

- *cache system*:

  - L2 (768 kB) = cache for the global memory
  - L1 (16 kB or 48 kB) = cache for each multiprocessor

  data from global memory is first loaded in L2, then in L1

  L1 cache and shared memory are implemented with the same circuits

  → two possible configurations: L1 on 16 kB and shared memory on 48 kB
  or
  L1on 48 kB and shared memory on 16 kB

- *registers in each core:*

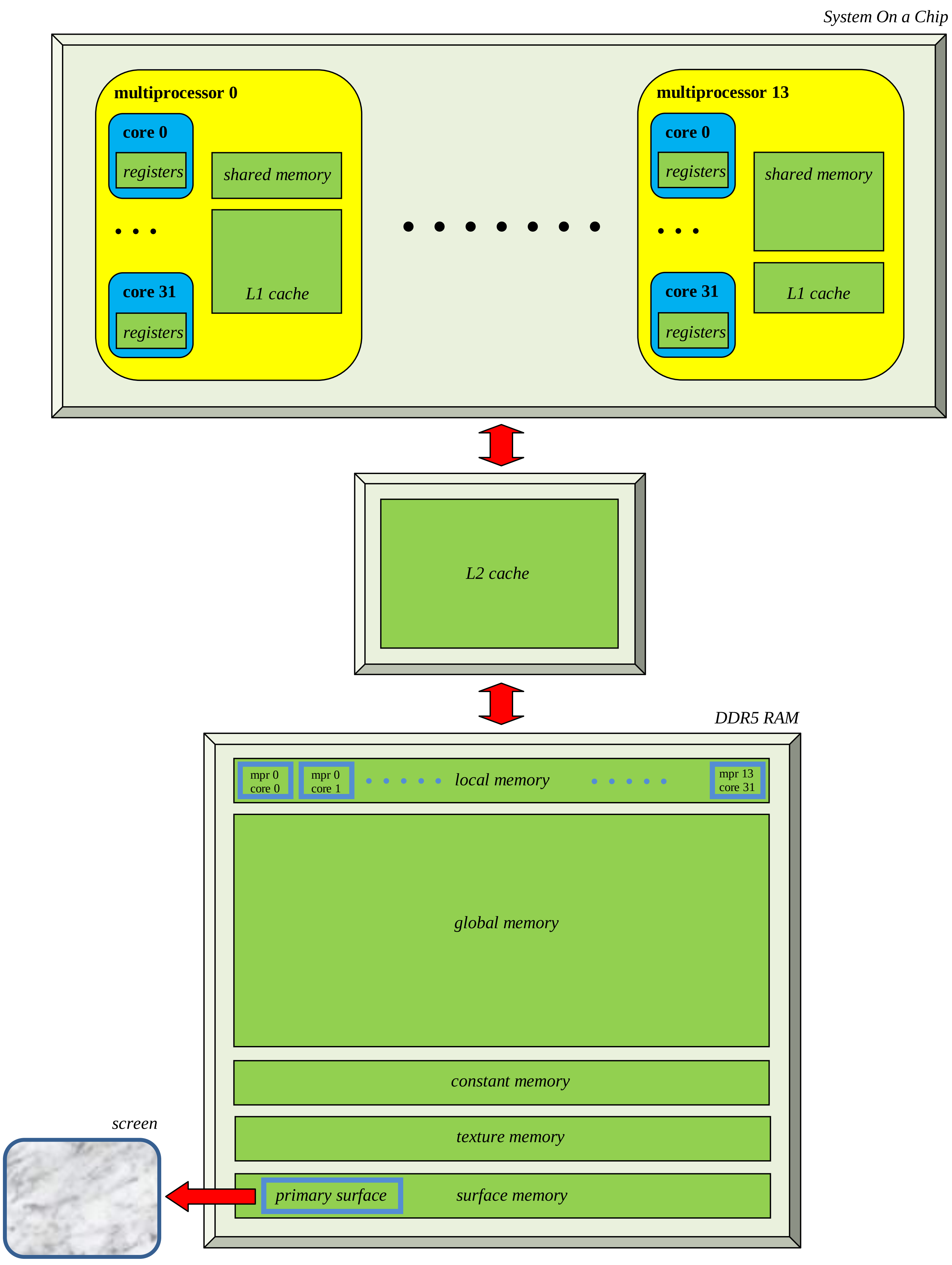
  = store arguments and local variables of functions executed on this core

  if not enough registers, the rest is stored in the *local memory* which is much slower

  → to avoid that, keep code executed on the cores as small as possible

- *constant memory*, *texture memory*, *surface memory*:

  - specialized memories useful for graphic applications
  - inside surface memory, the *primary surface* is used to refresh the screen


System On a Chip
multiprocessor 0
core 0
registers
shared memory
. . .
L1 cache
core 31
registers
multiprocessor 13
core 0
registers
shared memory
. . .
core 31
registers
L1 cache
L2 cache
DDR5 RAM
mpr 0 core 0
mpr 0 core 1
local memory
mpr 13 core 31
global memory
constant memory
texture memory
primary surface
surface memory
screen

execution model:

- SIMD: *Single Instruction Multiple Data* *example: vector supercomputers*

execution of instructions directly dictated by structure of data (vectors)

the same instruction is applied simultaneously over all data elements (vector elements)

→ only one CPU with only one instruction address counter, but with many arithmetic units

- SIMT: *Single Instruction Multiple Threads* = CUDA execution model

similar to SIMD but more flexible:

- can also execute "data-parallel code" like SIMD
- however, each core can execute the common code independently from the other cores

each core is an independent CPU and has its own instruction address counter

→ all cores may execute the same code but these executions eventually diverge
= different instructions executed at a given time

→ actually, some cores may execute different codes in the same time (up to 16 for 2.x)

warp: group of 32 threads executed simultaneously on the 32 cores of a multiprocessor

- MIMD: *Multiple Instructions Multiple Data*

several CPUs that execute simultaneously different codes on diverse data elements

- very often, MIMD is used with a single code executed by all the CPUs
(who has ever run hundreds of different codes concurrently ?)
- SIMT can also execute concurrently different codes on the cores

→ eventually, SIMT turns out to be quite similar to MIMD

CUDA programming model

data-based parallelism:

- to each basic element of *data* (pixel, matrix element)
  is associated a *thread* that will process this particular element

  = extremely fine granularity = *lightweight parallelism*

→ partition the data over two levels:

- *grid of blocks*
- *block of data elements* = *block of threads*

= this partition corresponds to the two levels of CUDA architecture:

block of threads ↔ multiprocessor	→ a block of threads is executed on a multiprocessor
thread ↔ core	→ a thread is executed on a core

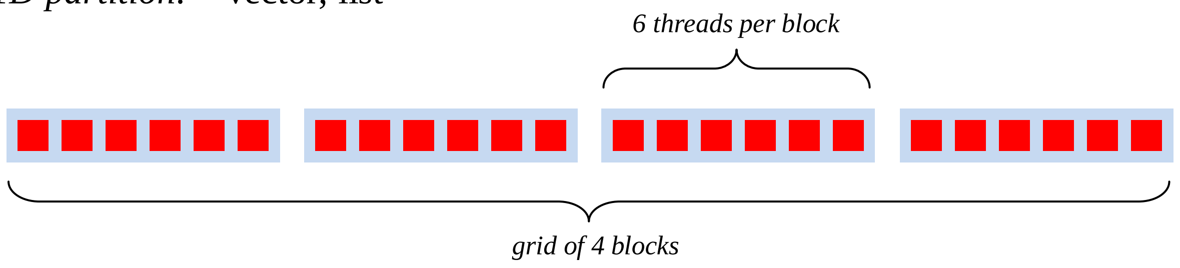
- inherent parallelism in the data should be sufficient to "overflow" the parallel architecture:
  - flow of blocks → dispatched over the multiprocessors
  - flow of threads in a block → dispatched over the cores of the multiprocessor

definition of the thread - data partition:

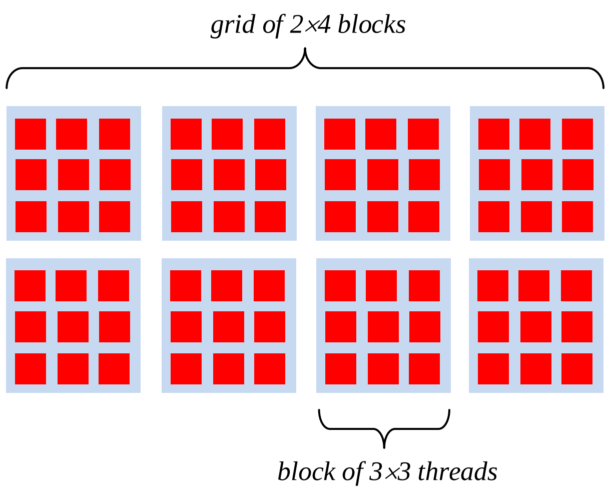
size of *grid of blocks* // size of *block of threads*

at most 1024 threads per block (for 2.0)

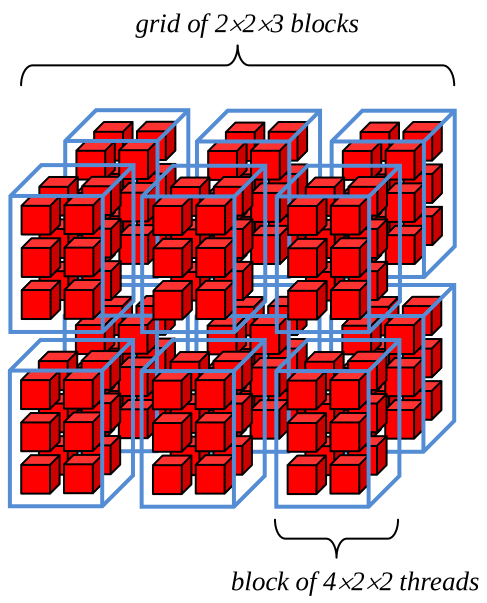
- *1D partition:* vector, list



- *2D partition:* matrix, picture (matrix of pixels)



- *3D partition:* volume (grid of voxels)



host code and device code:

the device (GPU) is just a coprocessor

→ the code running on the PC (*host code*) controls the code running on the device

- *device code* = *kernel* = code executed by each *thread* over each data element

  → executed in parallel over all the cores of all the multiprocessors

⚠ actually, up to 16 different kernels can run concurrently on all the cores (only for 2.x)

⚠ a kernel call should execute in less than 3 seconds, else it is interrupted by Windows OS

synchronization:

- *(non) synchronization between host and device:*

  by default, no synchronization between the host code and a kernel

  = the host code launches the kernel execution and continues its own execution

  but we may want the host code to wait for all the threads of the kernel to finish

  → define a *meeting point* inside the host code

- *synchronization between the threads of a block:*

  by default, all threads of a block execute independently from each other

  but we may want all of them to finish before the program can continue

  → define a *meeting point* inside the device code

- *synchronization between all the threads:*

  how to synchronize access by any threads to some shared data in global memory ?

  → use predefined *atomic operations*