

CUDA Programming

Outline

- GPU

- CUDA Introduction

 - What is CUDA

 - CUDA Programming Model

 - Advantages & Limitations

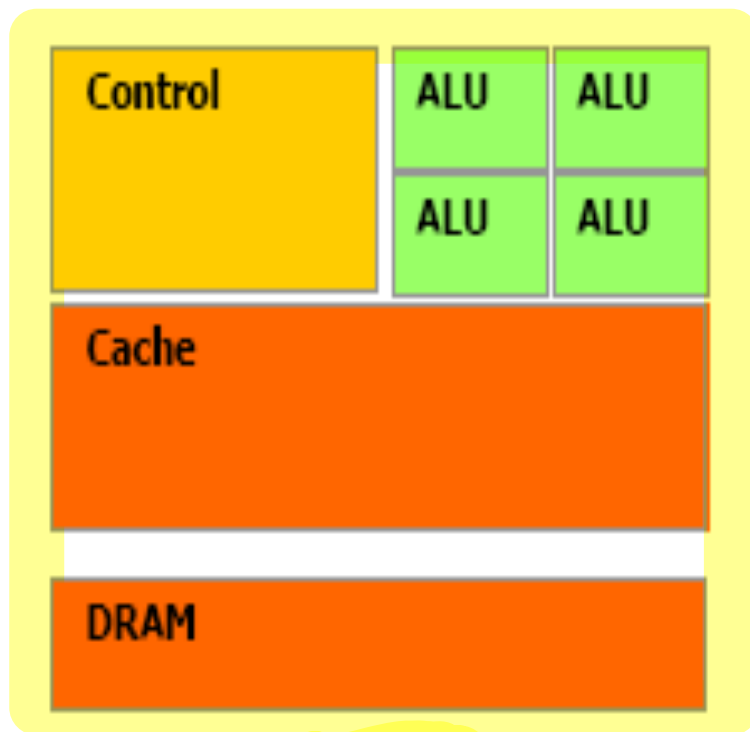
- CUDA Programming

- Future Work

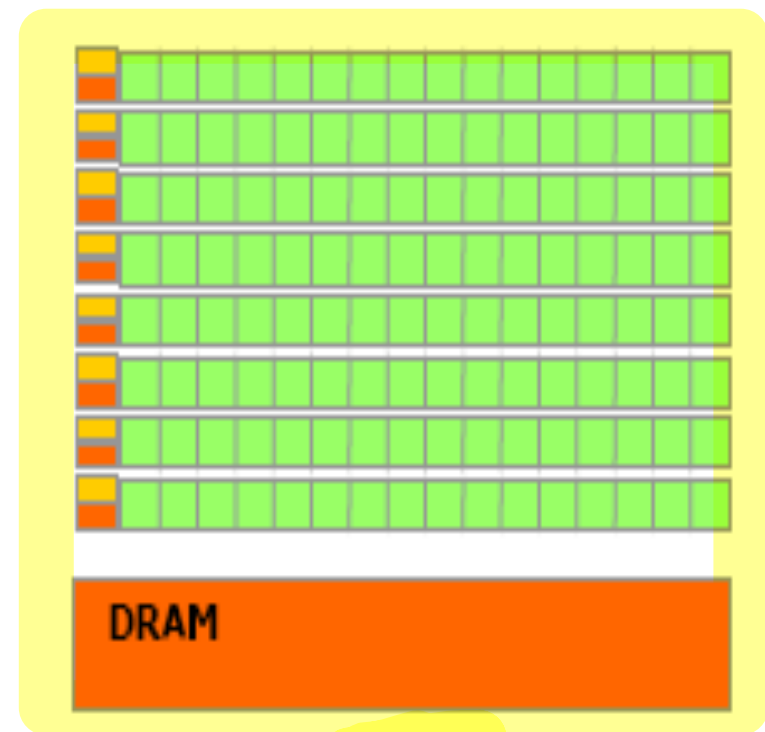
GPU

- ❑ GPUs are massively multithreaded many core chips
 - ❑ handle computation only for computer graphics
 - ❑ Hundreds of processors
 - ❑ Tens of thousands of concurrent threads
 - ❑ TFLOPs peak performance
 - ❑ Fine-grained data-parallel computation
 - ↳ data parallelism
- ❑ Users across science & engineering disciplines are achieving tenfold and higher speedups on GPU

CPU v/s GPU

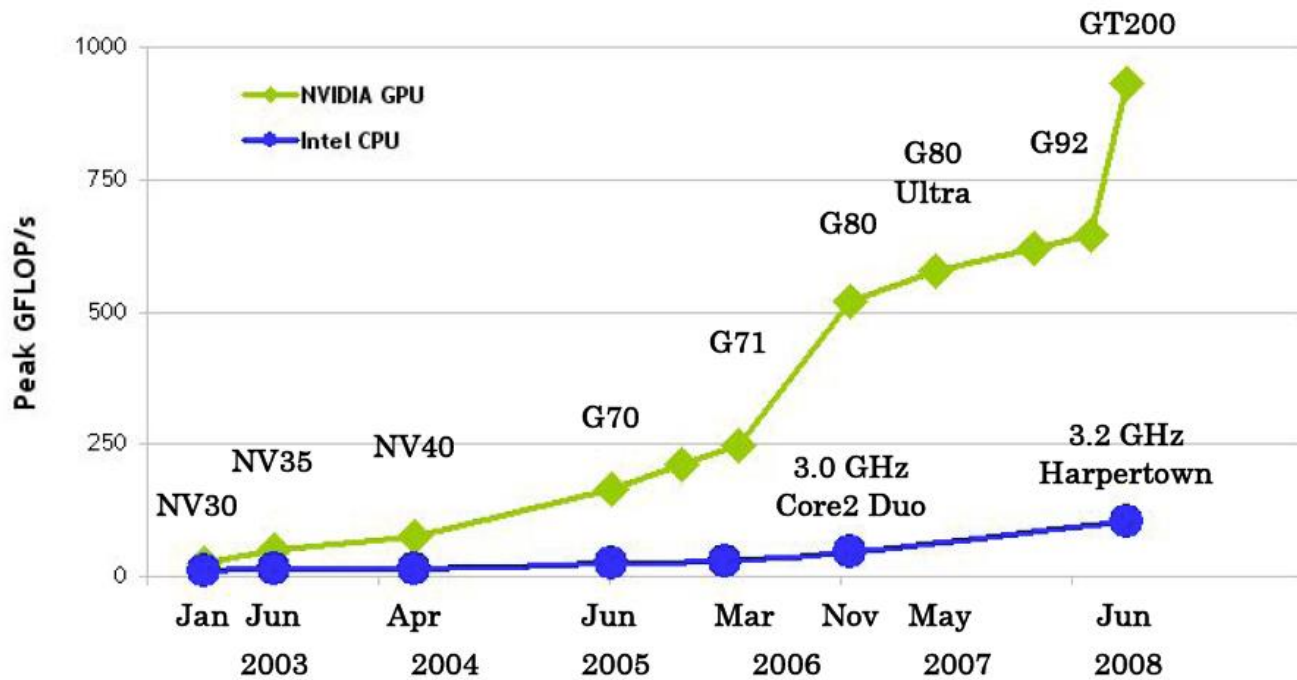


CPU



GPU

CPU v/s GPU



GT200 = GeForce GTX 280

G71 = GeForce 7900 GTX

NV35 = GeForce FX 5950 Ultra

G92 = GeForce 9800 GTX

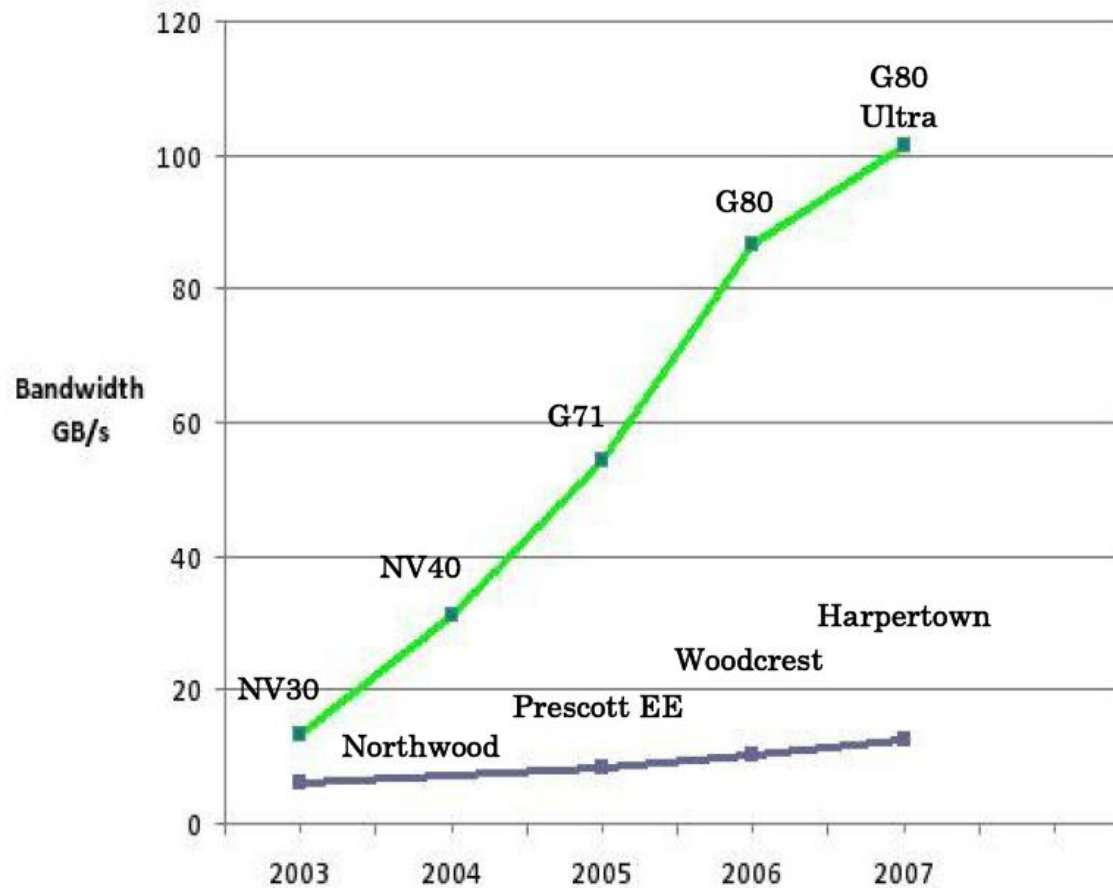
G70 = GeForce 7800 GTX

NV30 = GeForce FX 5800

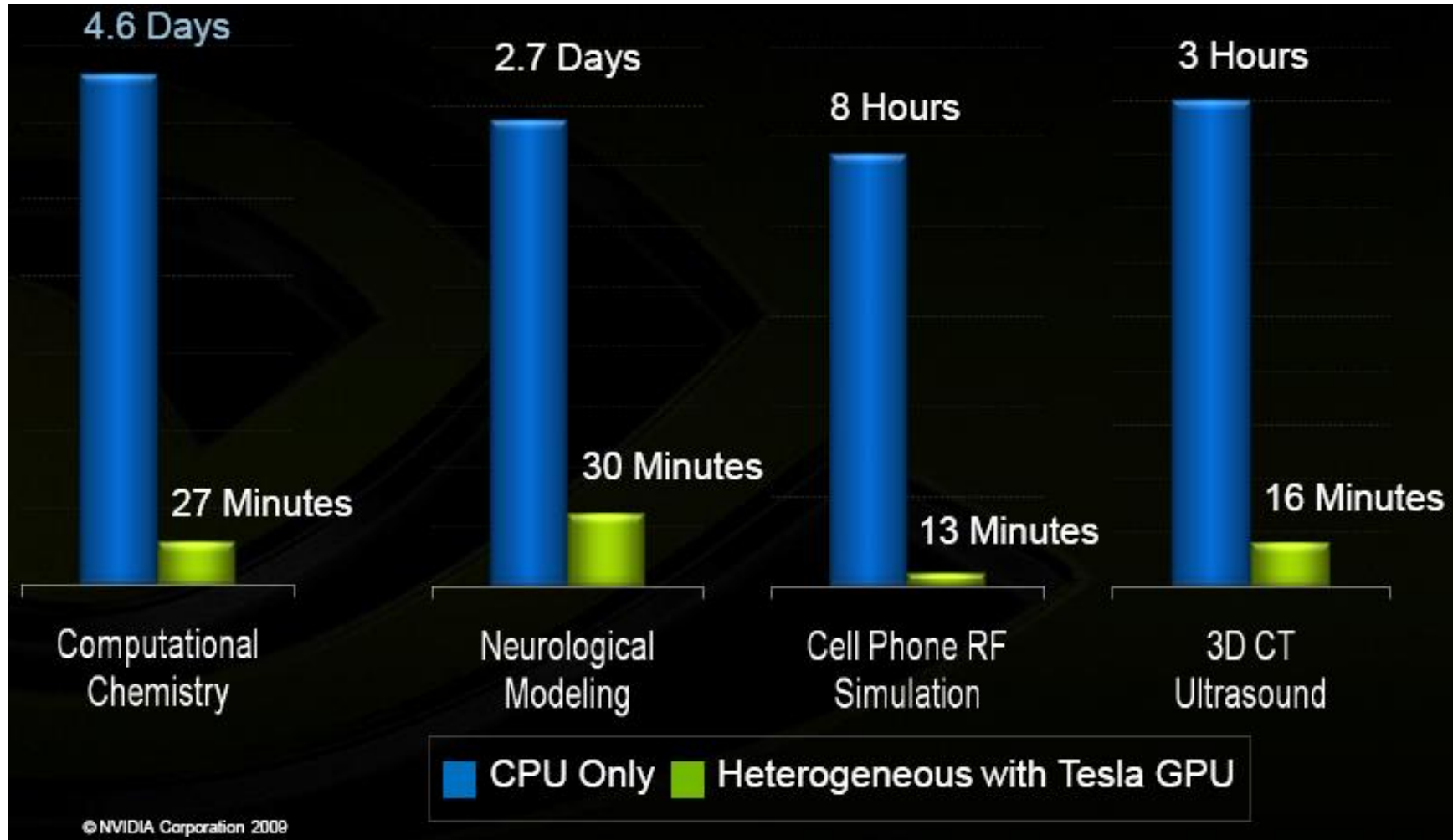
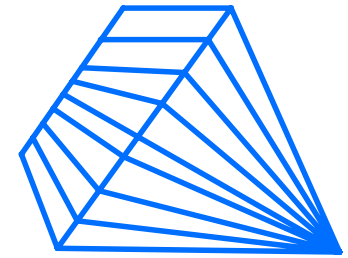
G80 = GeForce 8800 GTX

NV40 = GeForce 6800 Ultra

CPU v/s GPU



CPU v/s GPU



GPGPU

GPU في الأصل للرسومات
GPU في كل الاستخدامات
التي إذا كان الكود Parallell

- What is GPGPU?
 - General purpose computing on GPUs
 - GPGPU is the use of a GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).
- Why GPGPU?
 - Massively parallel computing power
 - Inexpensive

GPGPU

- How?

- CUDA

- OpenCL

- DirectCompute

frameworks



What is CUDA?

- ❑ CUDA is the acronym for Compute Unified Device Architecture.
 - ❑ A parallel computing architecture developed by NVIDIA.
 - ❑ Heterogeneous serial-parallel computing
 - ❑ The computing engine in GPU.
 - ❑ CUDA can be accessible to software developers through industry standard programming languages.
- ❑ CUDA gives developers access to the instruction set and memory of the parallel computation elements in GPUs.

Heterogeneous Computing

- Terminology:

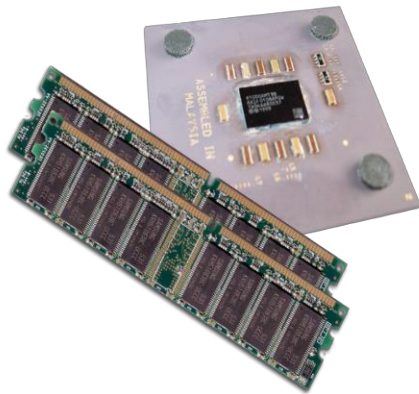
- **Host** The CPU and its memory (host memory)

- **Device** The GPU and its memory (device memory)

cpu
GPU

GPU Device , CPU host

في الـ GPU
الـ Device



Host



Device

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE * 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2 * RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2 * RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2 * RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>>(d_in + RADIUS,
    d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

serial code

parallel code

serial code

CPU



GPU



CPU



CUDA Kernels and Threads

- ❑ Parallel portions of an application are executed on the device as kernels

cod that run in parallel

- ❑ **A Kernel** is a Function that runs on a device
 - ❑ One kernel is executed at a time
 - ❑ Many threads execute each kernel

- ❑ Differences between CUDA and CPU threads

- ❑ CUDA threads are extremely lightweight

- ❑ Very little creation overhead

- ❑ Instant switching

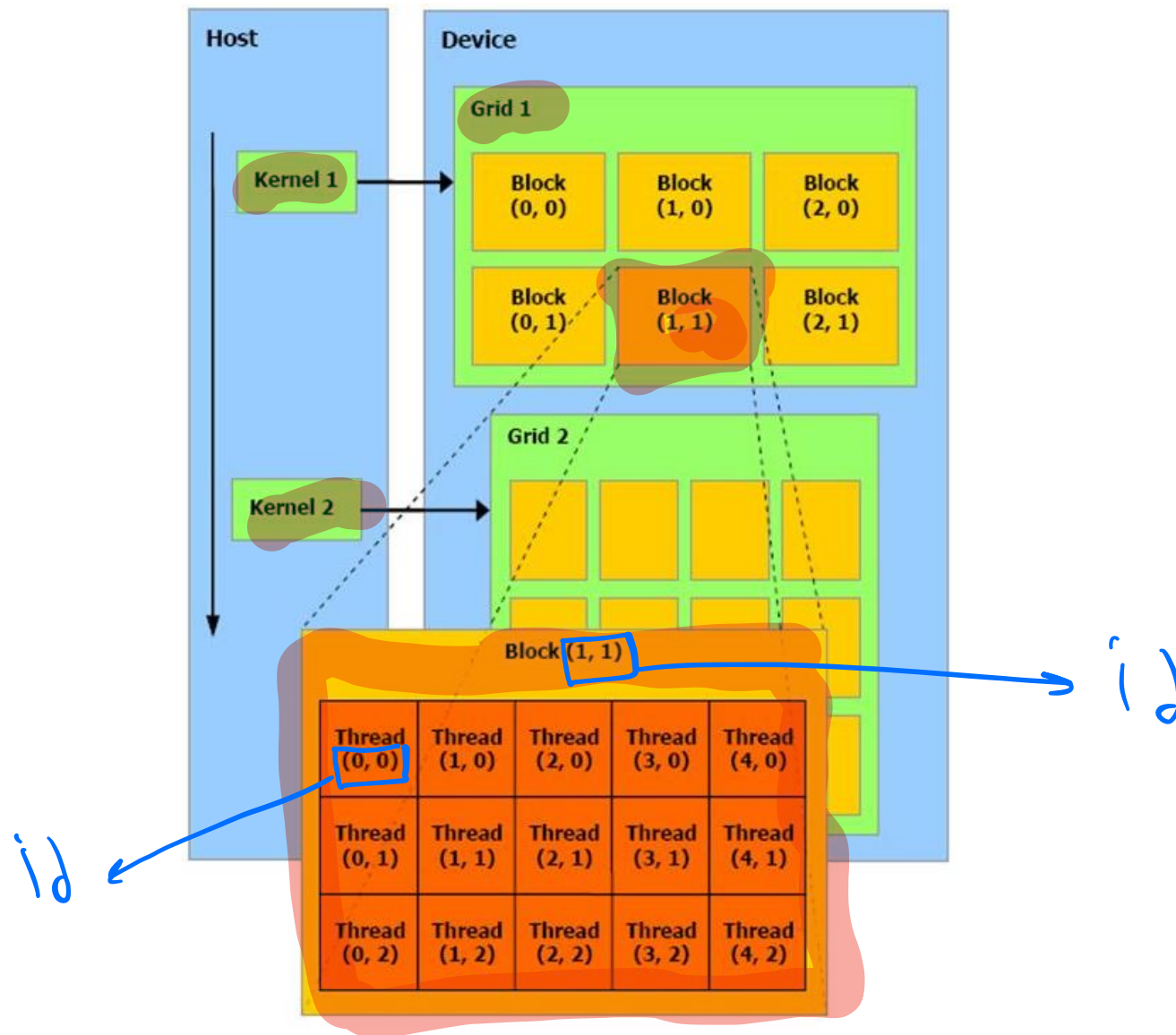
المهم في الانشاء

CUDA Programming Model

- ❑ A kernel is executed by a grid of thread blocks
- ❑ A thread block is a batch of threads that can cooperate with each other by:
 - ❑ Sharing data through shared memory
 - ❑ Synchronizing their execution
- ❑ Threads from different blocks cannot cooperate

* threads from same block can cooperate

CUDA Programming Model



CUDA Programming Model

- All threads within a block can
 - Share data through 'Shared Memory'
 - Synchronize using '_syncthreads()'

امر! اذا وصل له ال thread سينتظر ال thread الباقين يخلصون

- Threads and Blocks have unique IDs
 - Available through special variables

كندهم لا عشان يعرف كل واحد أي شي ال data الي كنده

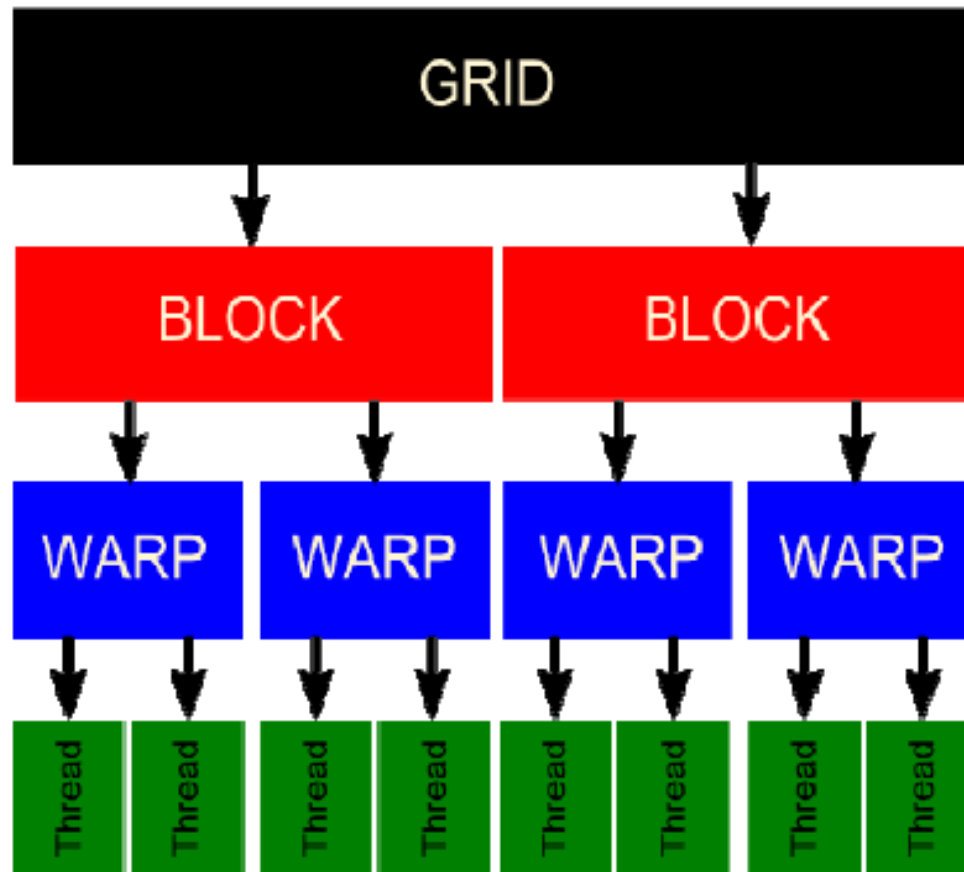
CUDA Programming Model

- SIMT (Single Instruction Multiple Threads) Execution

warp \Rightarrow 32 thread

- Threads run in groups of 32 called warps
- Every thread in a warp executes the same instruction at a time \Rightarrow SIMT

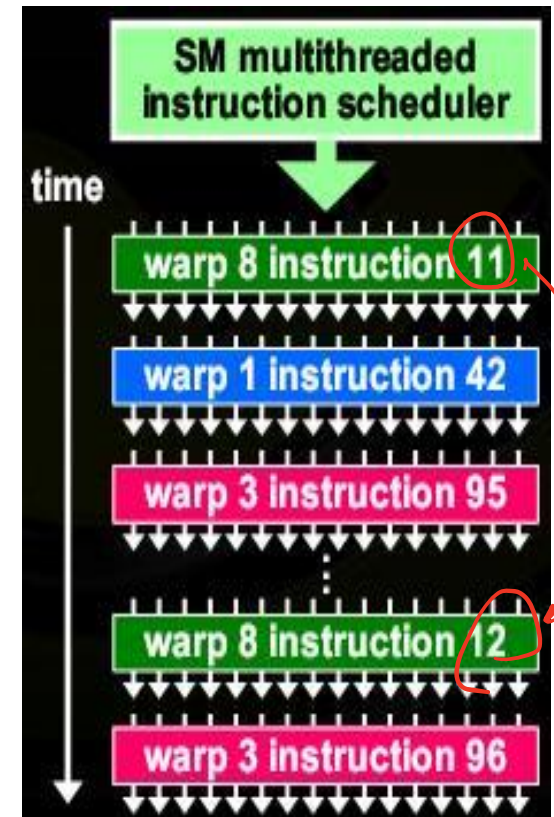
CUDA Programming Model



CUDA Programming Model

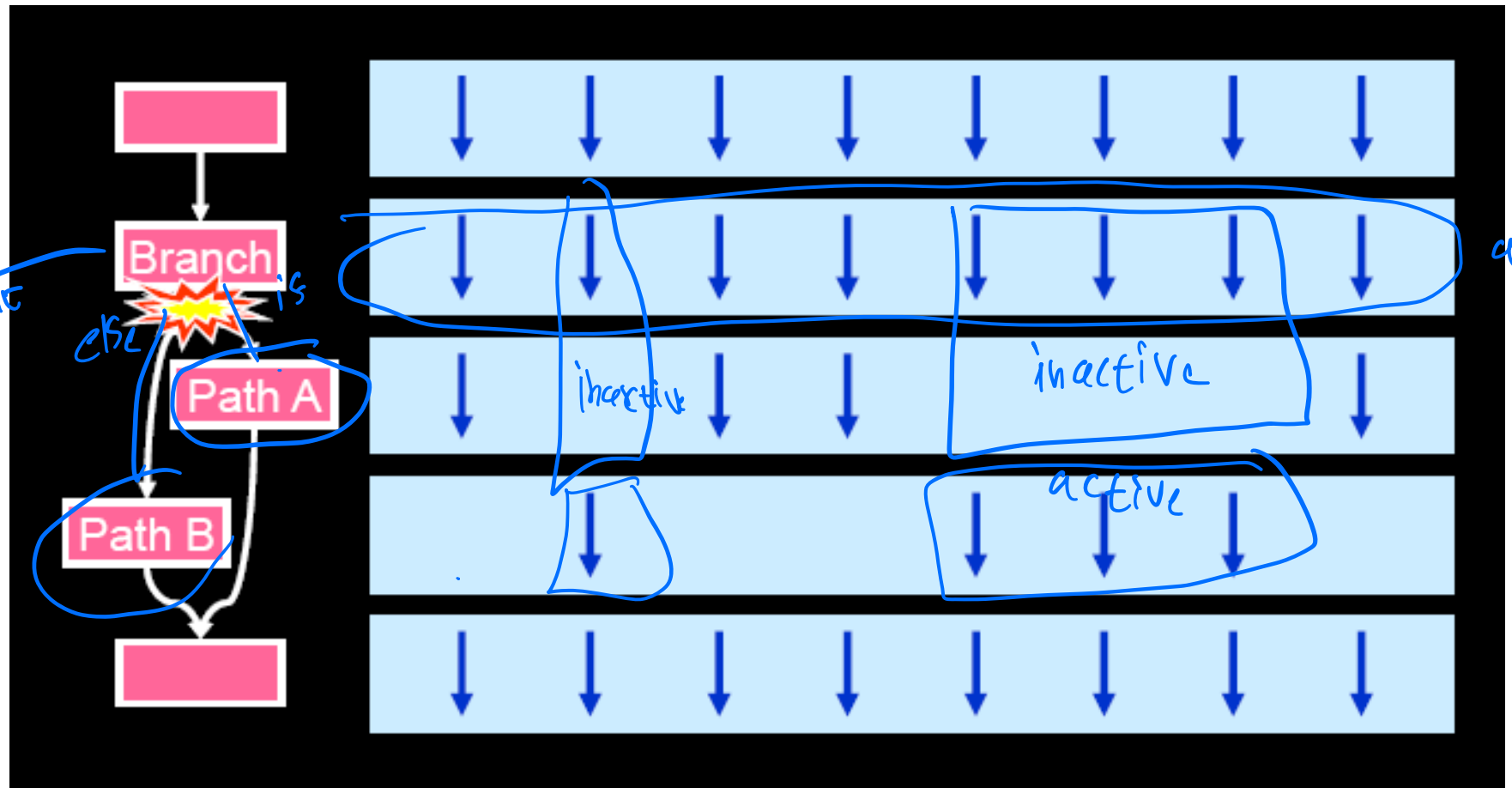
Single Instruction Multiple Thread (SIMT) Execution:

- Groups of 32 threads formed into **warps**
 - always executing same instruction
 - share instruction fetch/dispatch
 - some become **inactive** → اذا فيه if/else
 - when code path diverges
 - hardware **automatically handles divergence**
- **Warps** are primitive unit of scheduling
 - all warps from all active blocks are time-sliced



عمل warp يماند حقه من الوقت

Control Flow Divergence



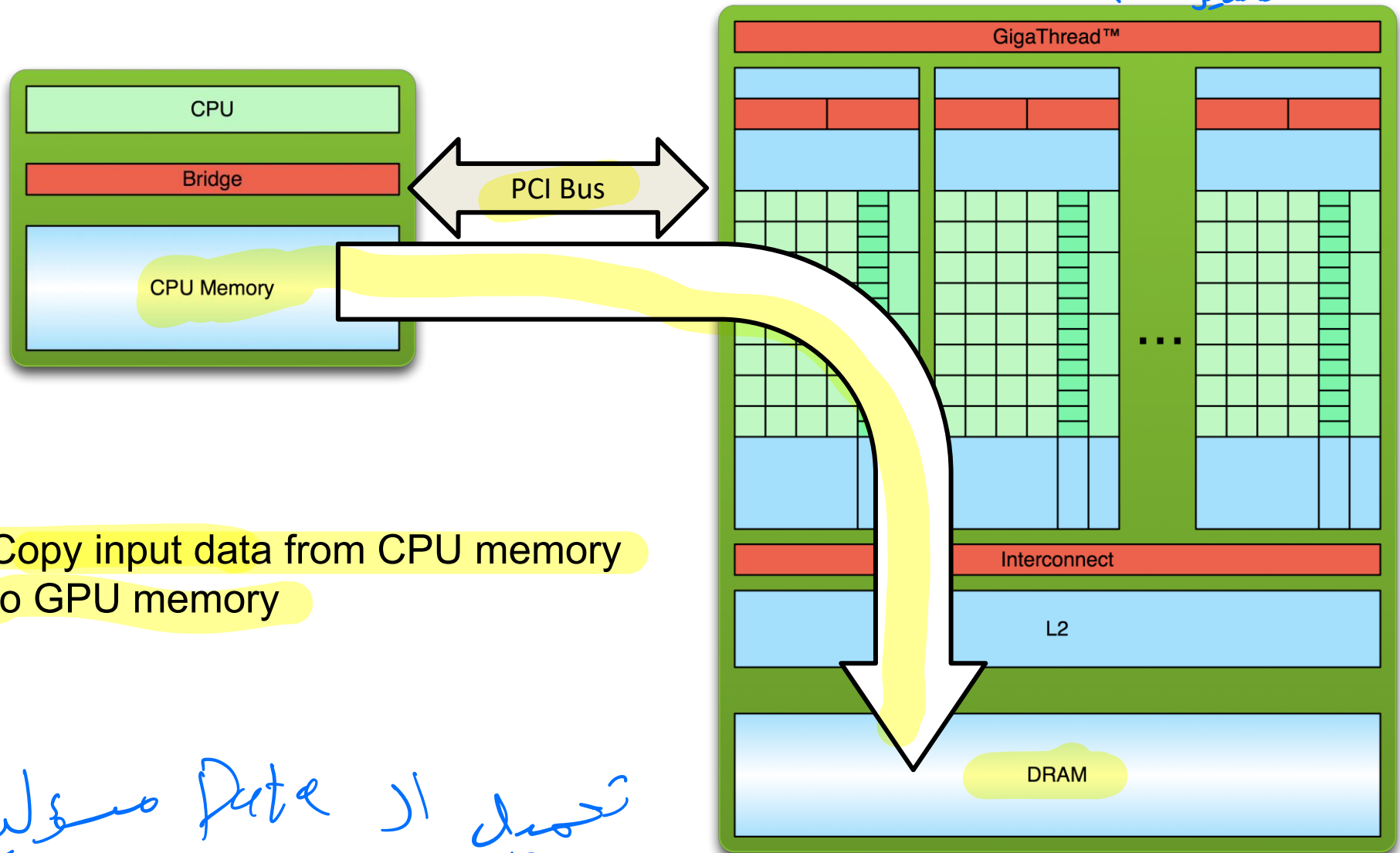
Courtesy Fung et al. MICRO '07

Courtesy Fung et al. MICRO-07

كل الـ thread اذ 32
يشتغلون معاً
لكل وحدة الأعداد

الـ thread الباغيين ما ينتفرون يعني
الـ threads متساوي الـ كذا (pathA) باقي الـ thread اي مفرد من الوحدات

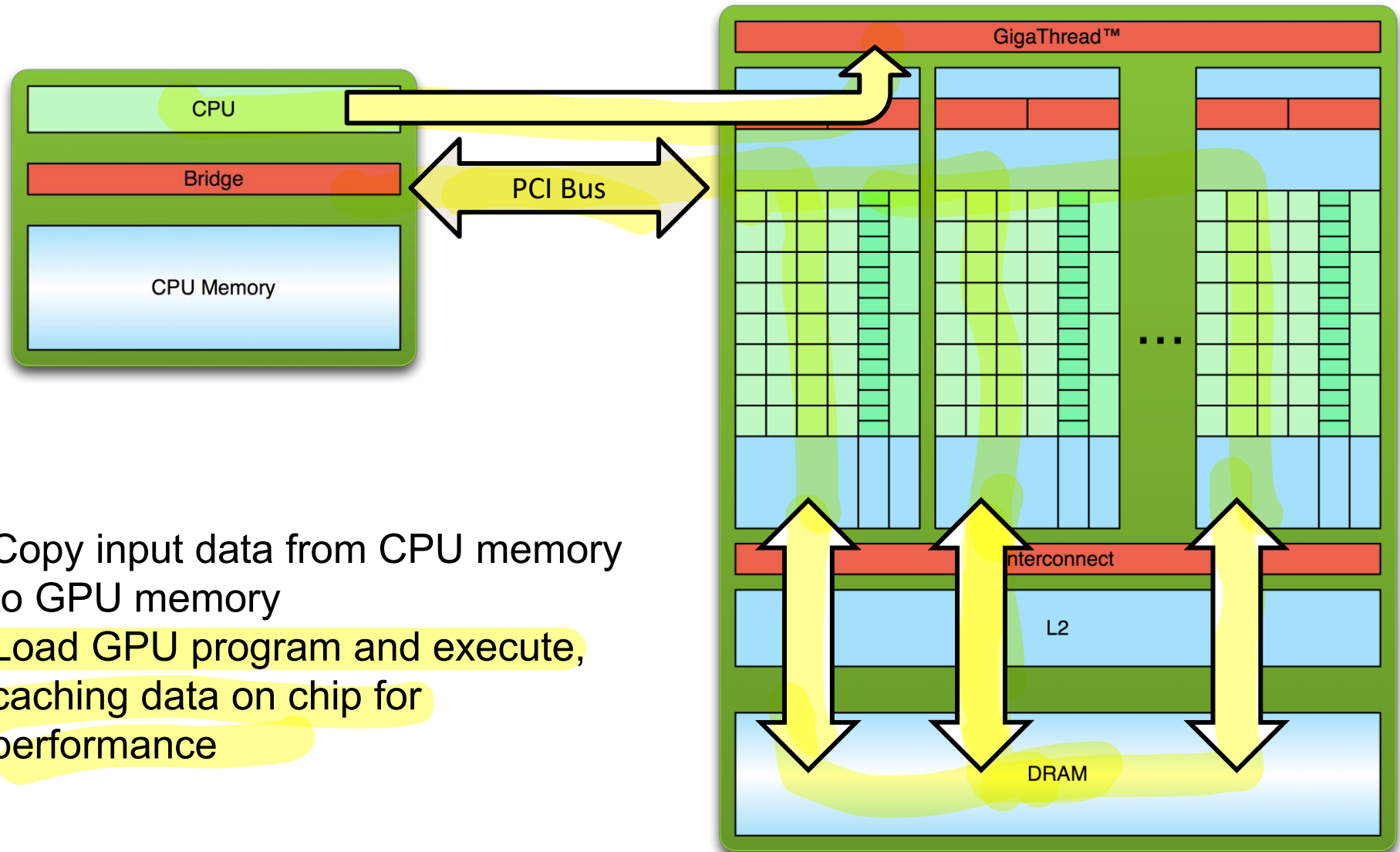
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory

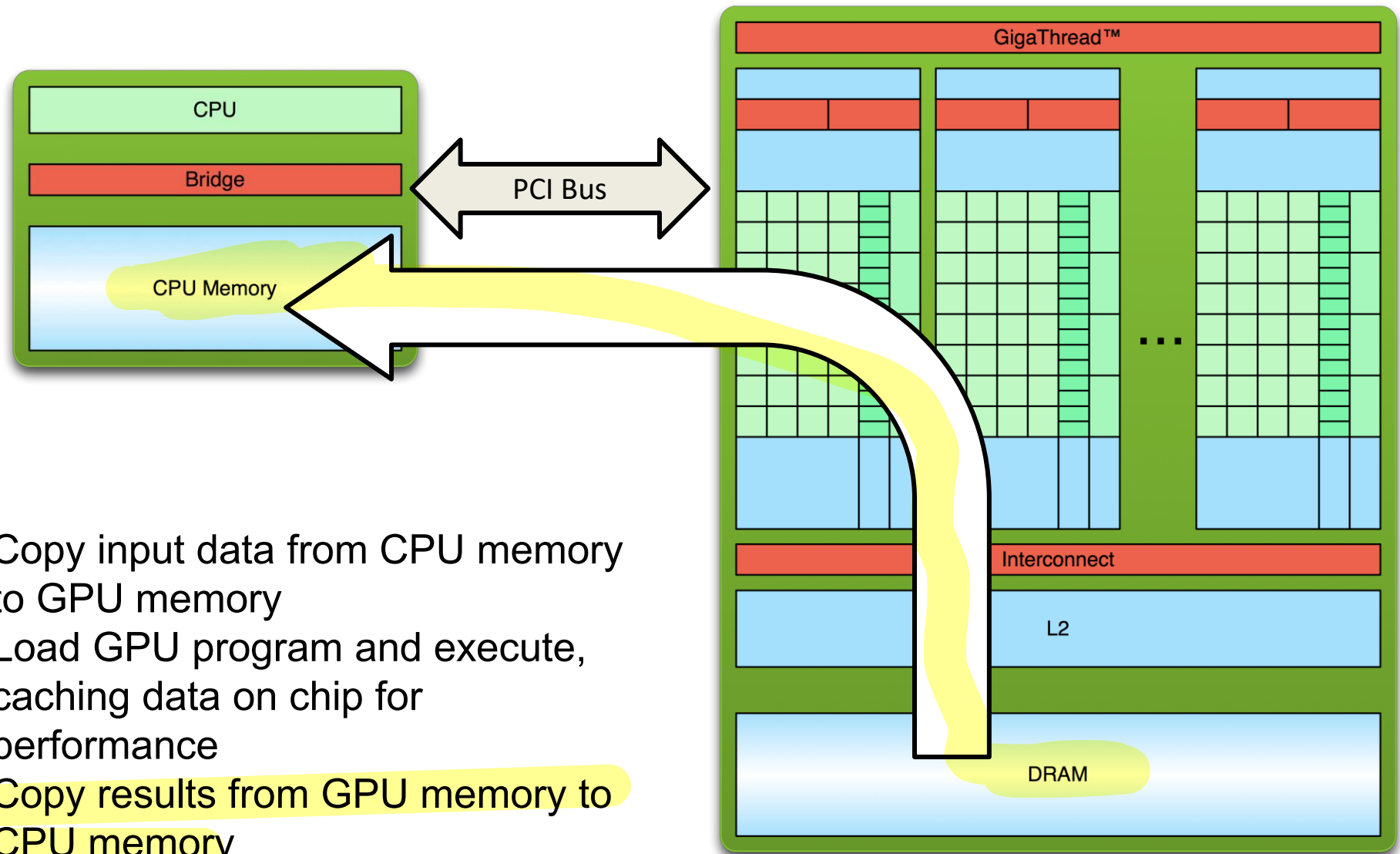
تحويل Data مسؤولية المبرمج

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow

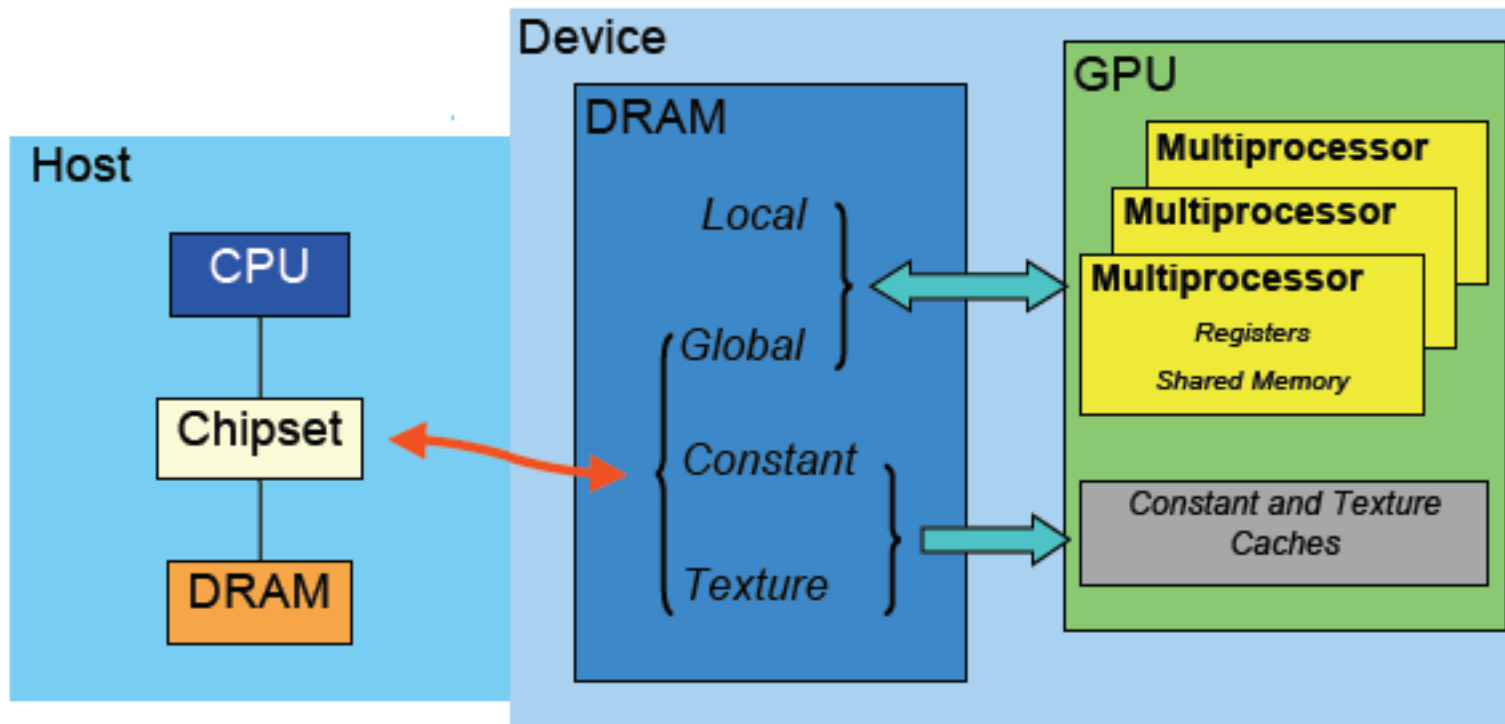


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Memory Model

- Types of device memory
 - **Registers** – read/write per-thread
 - **Local Memory** – read/write per-thread
 - **Shared Memory** – read/write per-block
 - **Global Memory** – read/write across grids
 - **Constant Memory** – read across grids
 - **Texture Memory** – read across grids

Memory Model

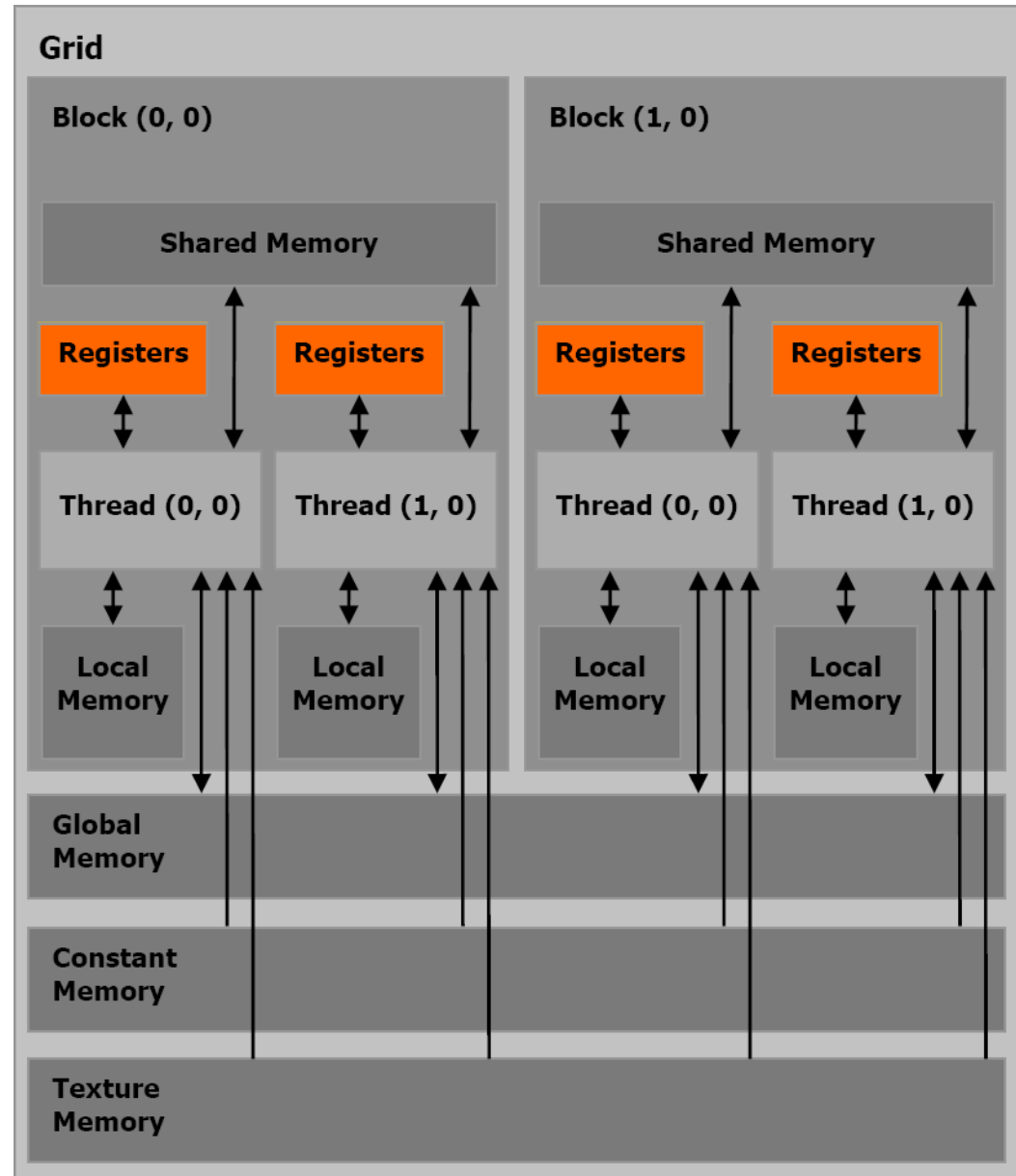


Memory Model

There are 6 Memory Types :

- **Registers**

- on chip
- fast access
- per thread
- limited amount
- 32 bit

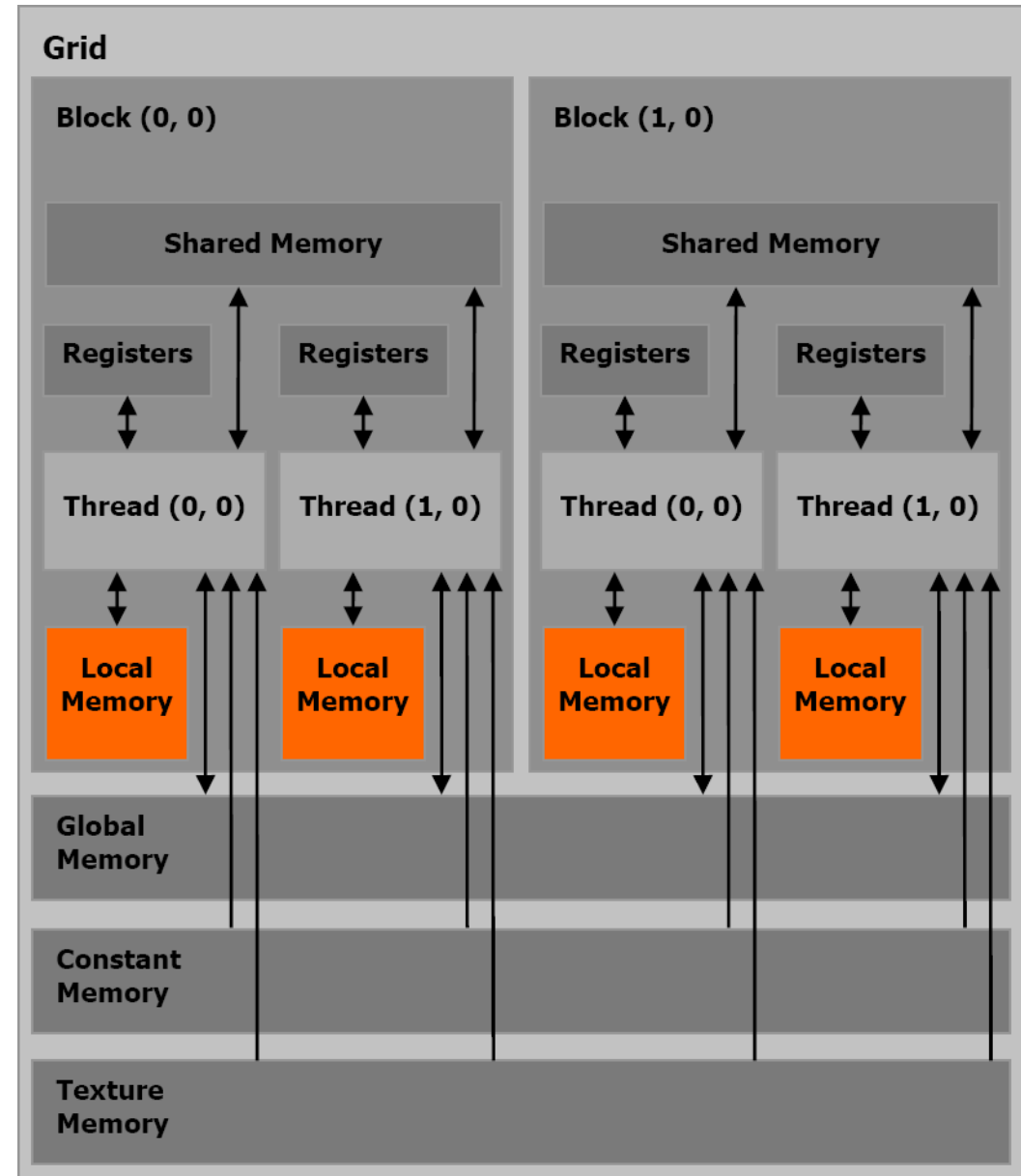


Memory Model

There are 6 Memory Types :

- Registers
- **Local Memory**
 - in DRAM
 - slow
 - non-cached
 - per thread
 - relative large

local variables ذخیره

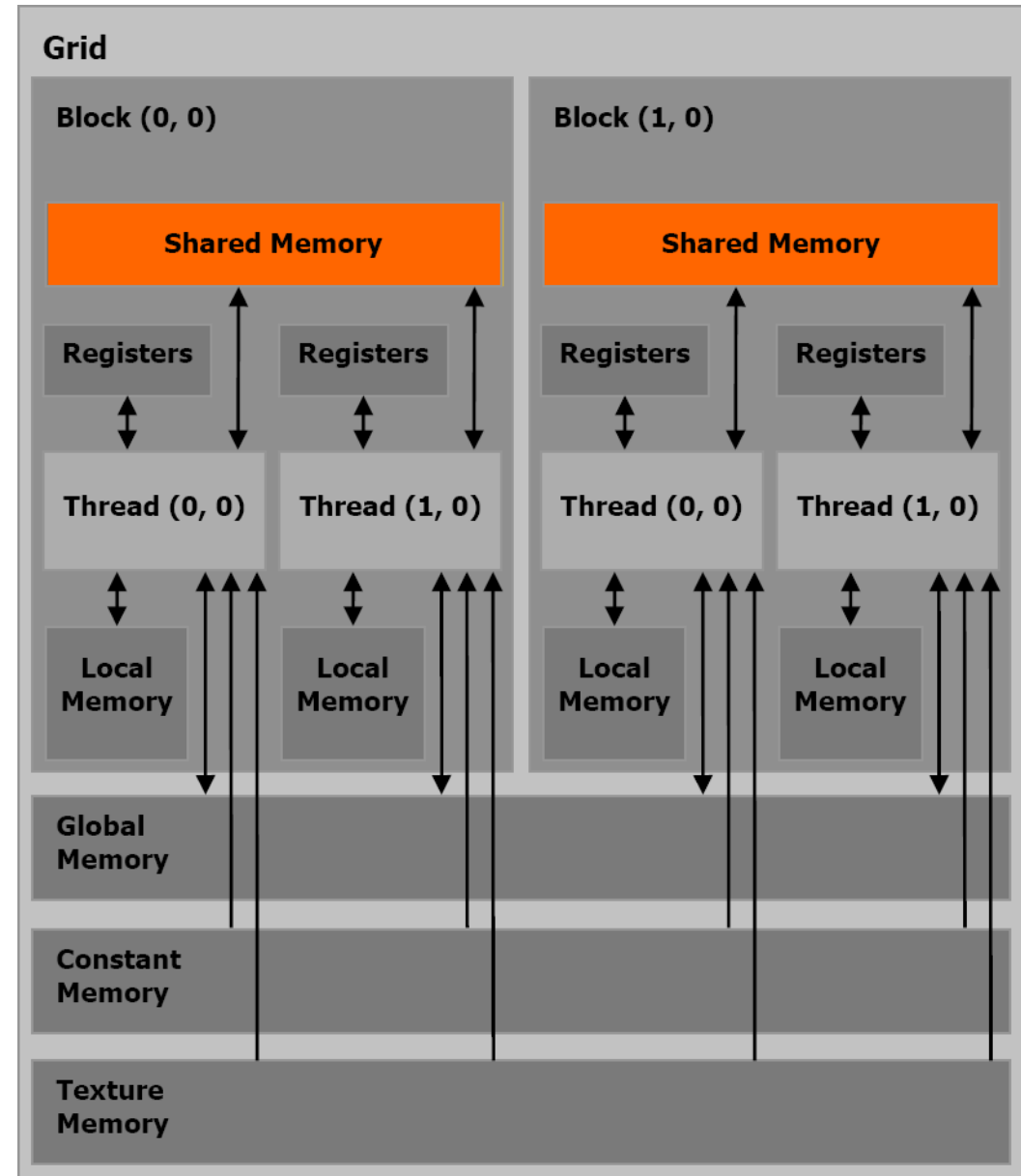


Memory Model

There are 6 Memory Types :

- Registers
- Local Memory
- **Shared Memory**
 - on chip
 - fast access
 - per block
 - 16 KByte
 - synchronize between threads

share memory in same thread in same block only



Memory Model

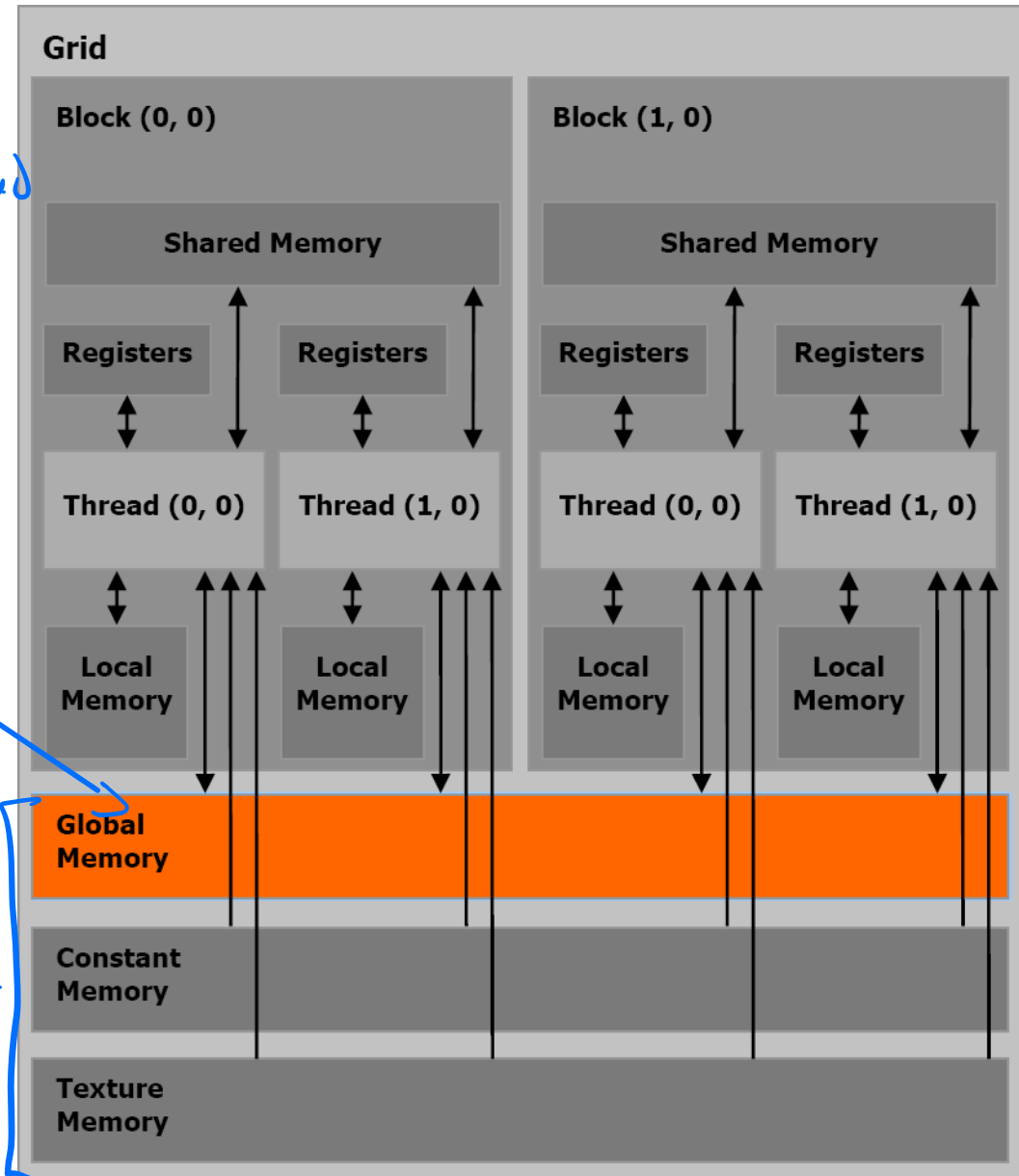
There are 6 Memory Types :

- Registers
- Local Memory
- Shared Memory
- **Global Memory**
 - in DRAM
 - slow
 - non-cached
 - per grid
 - communicate between grids

shared with all
thread

because
will be overhead

DRAM



Memory Model

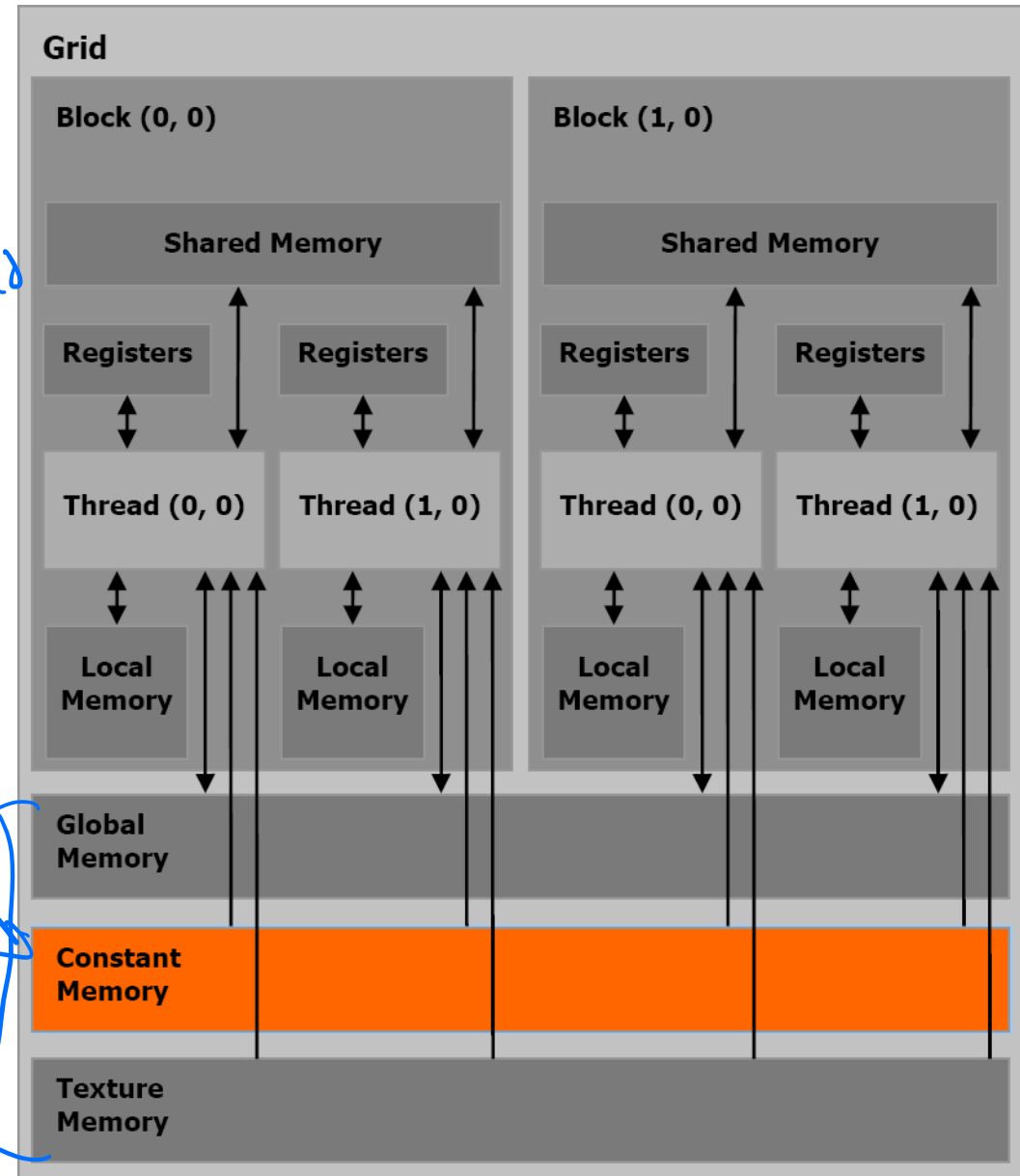
There are 6 Memory Types :

- Registers
- Local Memory
- Shared Memory
- Global Memory
- **Constant Memory**

- in DRAM
- cached
- per grid
- read-only

*because
no overhead*

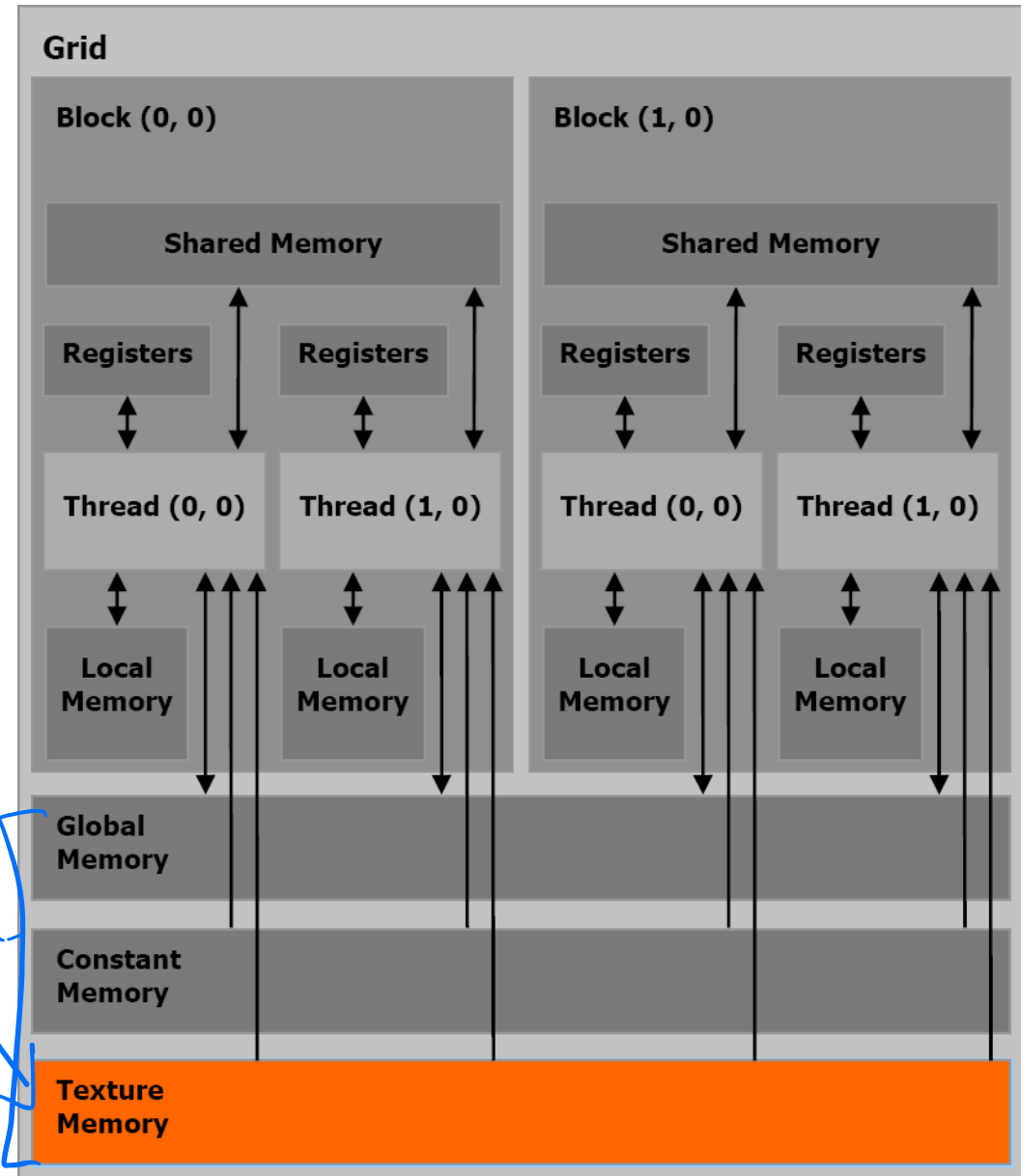
DRAM



Memory Model

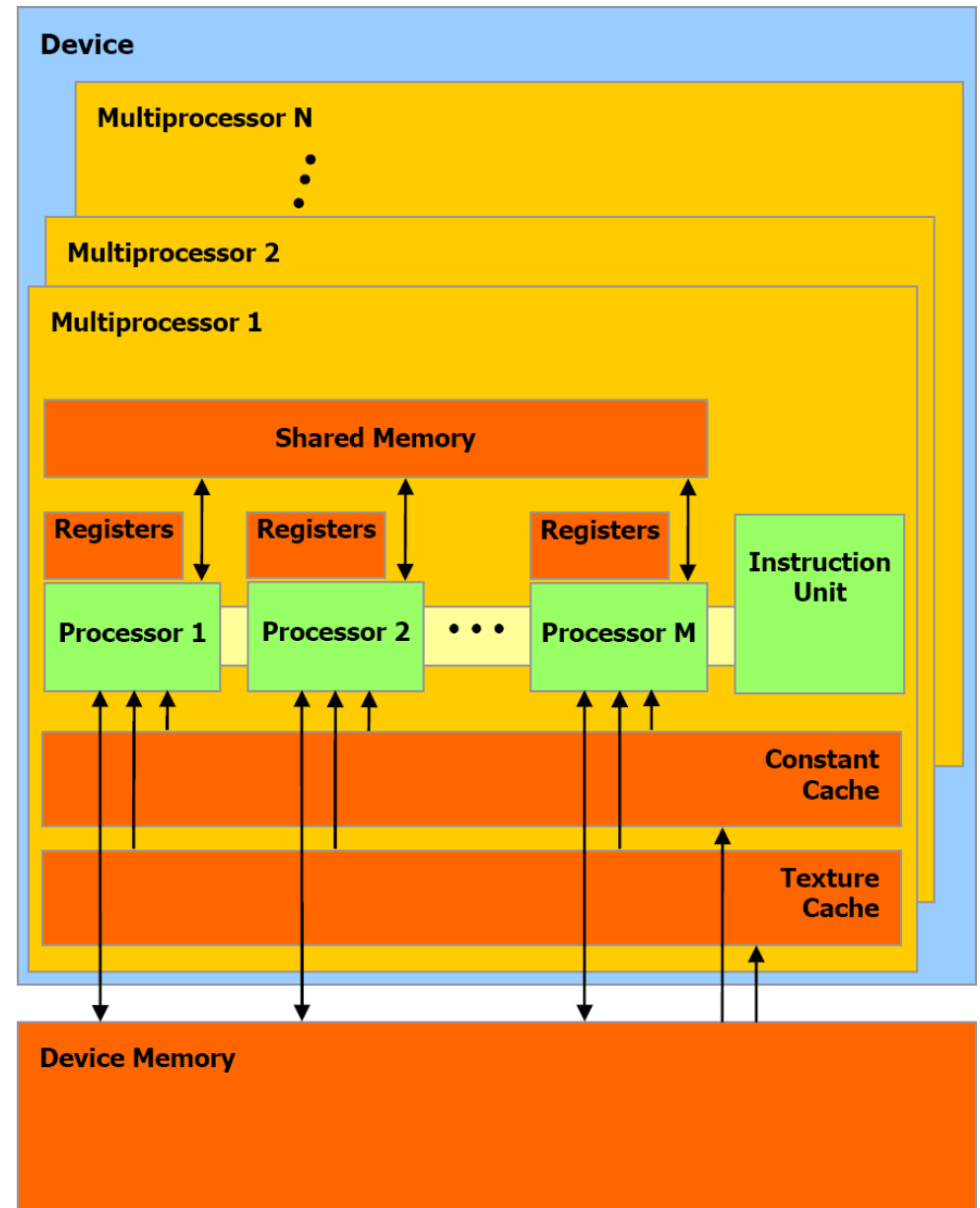
There are 6 Memory Types :

- Registers
- Local Memory
- Shared Memory
- Global Memory
- Constant Memory
- **Texture Memory**
 - in DRAM
 - cached
 - per grid
 - read-only



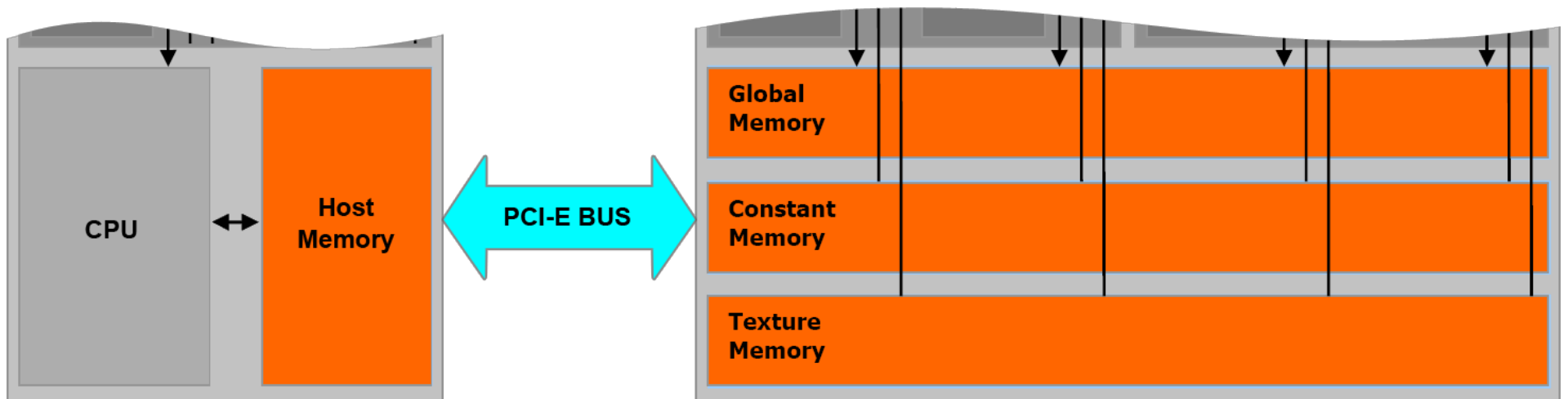
Memory Model

- Registers
- Shared Memory
 - on chip
- Local Memory
- Global Memory
- Constant Memory
- Texture Memory
 - in Device Memory



Memory Model

- Global Memory
- Constant Memory
- Texture Memory
 - managed by host code
 - persistent across kernels



Advantages of CUDA

- ❑ **CUDA has several advantages over traditional general purpose computation on GPUs:**
 - ❑ **Scattered reads** – code can read from arbitrary addresses in memory.
 - ❑ **Shared memory** - CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads.

Limitations of CUDA

- ❑ **CUDA has several limitations over traditional general purpose computation on GPUs:**
 - ❑ **A single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.**
 - ❑ **The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.**
 - ❑ **CUDA-enabled GPUs are only available from NVIDIA.**