Introduction to parallel programming using CUDA

Hardware & Software

Albert García García

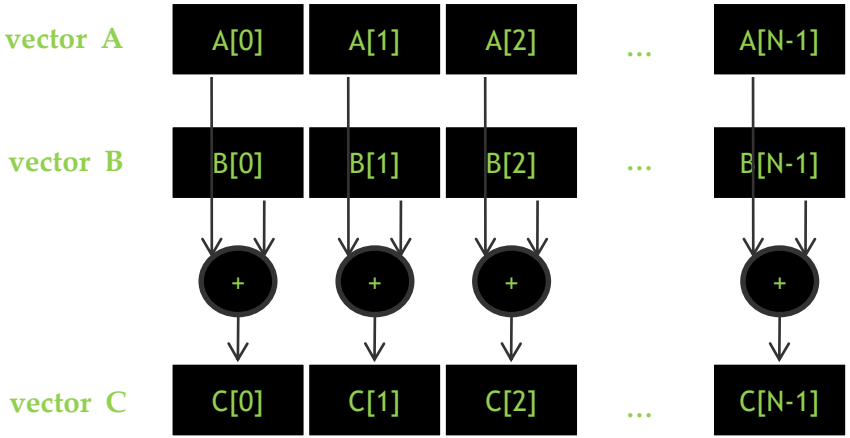
agarcia @ dtic.ua.es

Objectives

- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To learn about CUDA architecture (Software/Hardware)
- To learn the basic API functions in CUDA host code
  - Device Memory Allocation
  - Host-Device Data Transfer
- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
  - Hierarchical thread organization
  - Launching parallel execution
  - Thread index to data index mapping
- To learn the basic concepts involved in a simple CUDA kernel function
  - Declaration
  - Built-in variables
  - Thread index to data index mapping

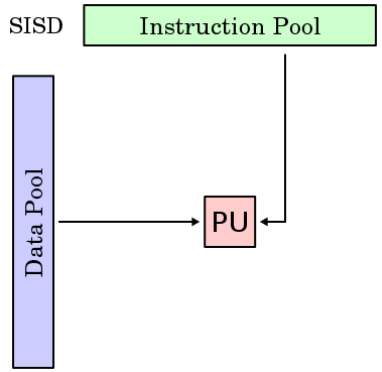
Parallelism (computing)

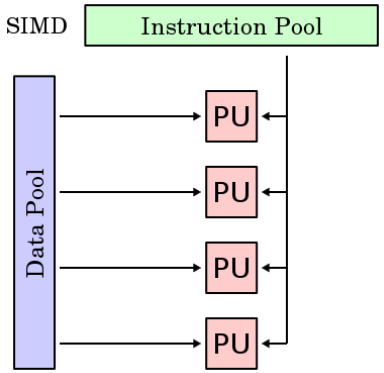
- Leveraging parallelism is a form of computing in which several calculations can be performed simultaneously.
- Based on the principle of dividing large problems into several small problems, which are then solved in parallel. (Divide and conquer)

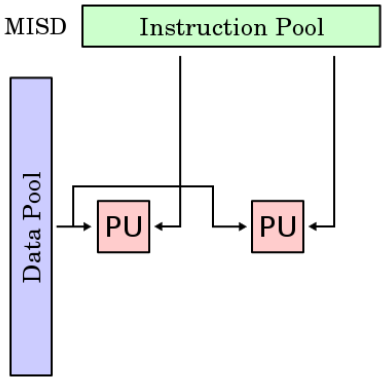


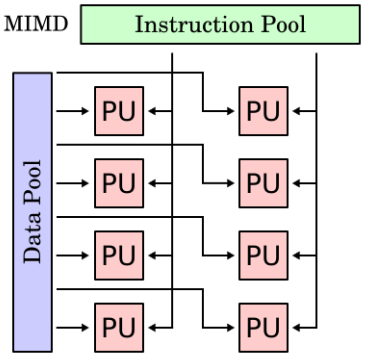
Computer Architecture

Flynn's taxonomy



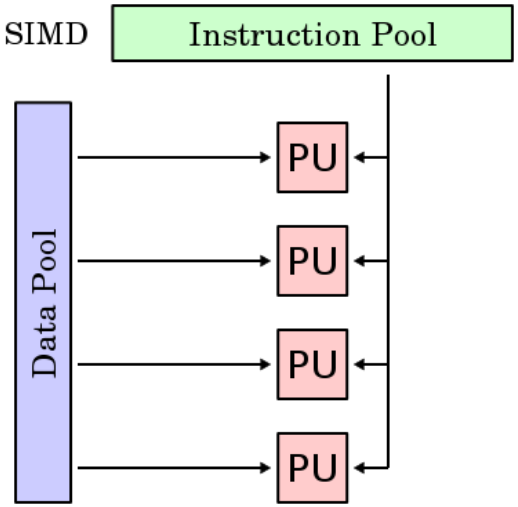






Computer Architecture

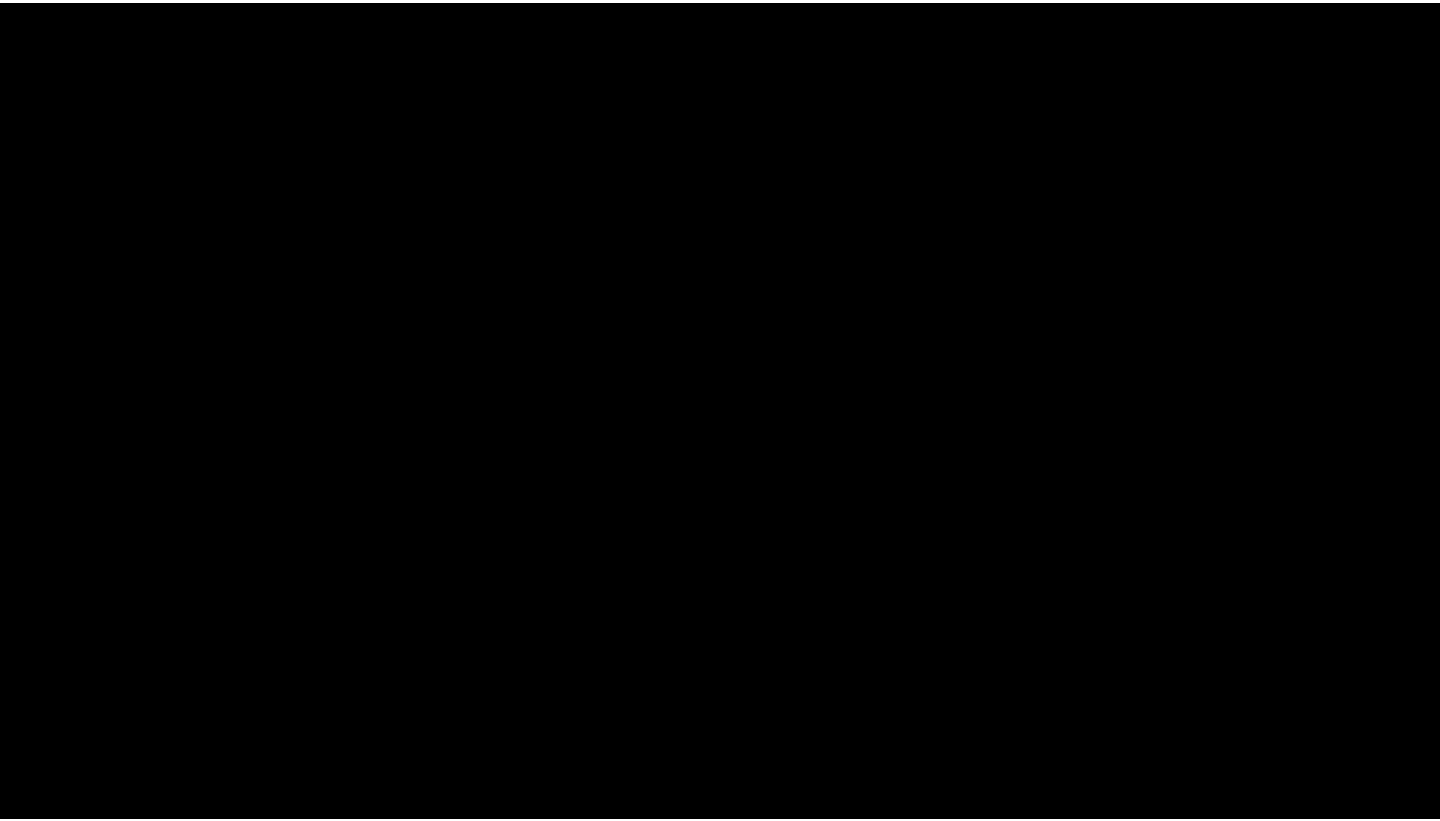
SIMD



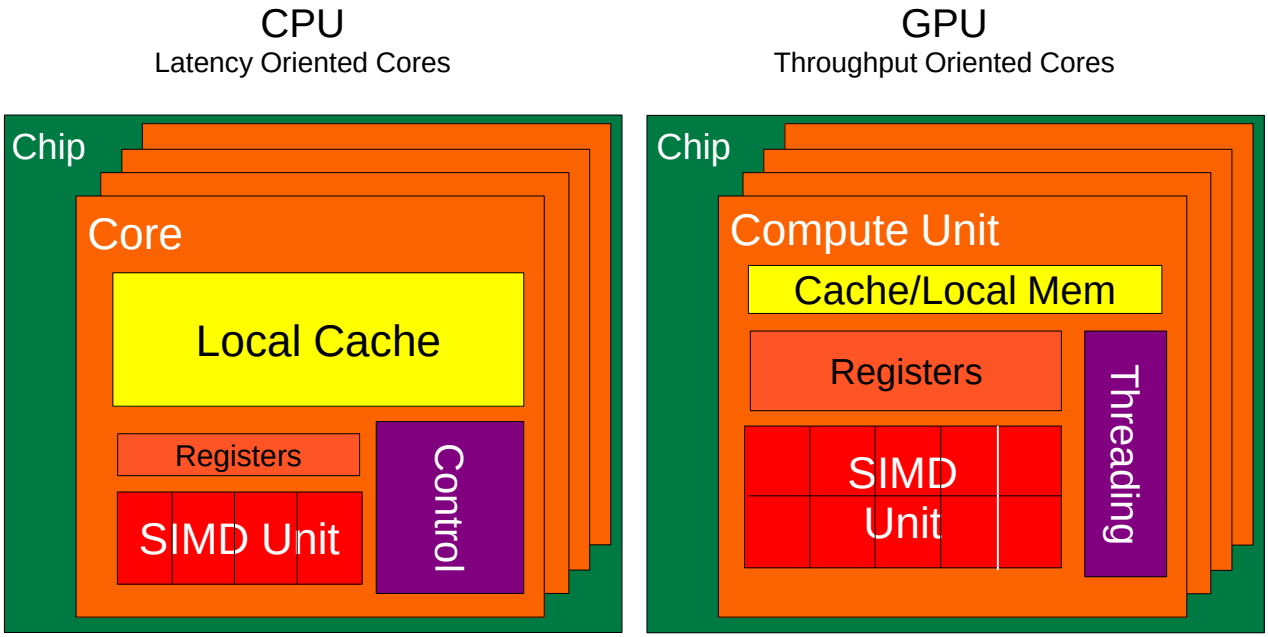
- **S**ingle **I**nstruction **M**ultiple **D**ata
  - Instruction operates on all loaded data in a single operation
  - The data is understood to be in blocks, and a number of values can be loaded all at once
  - Example: Intel's AVX SIMD instructions now process 256 bits of data at once

SIMD instructions are widely used to process 3D graphics, although modern graphics cards with embedded SIMD (a.k.a SIMT) have largely taken over this task from the CPU.

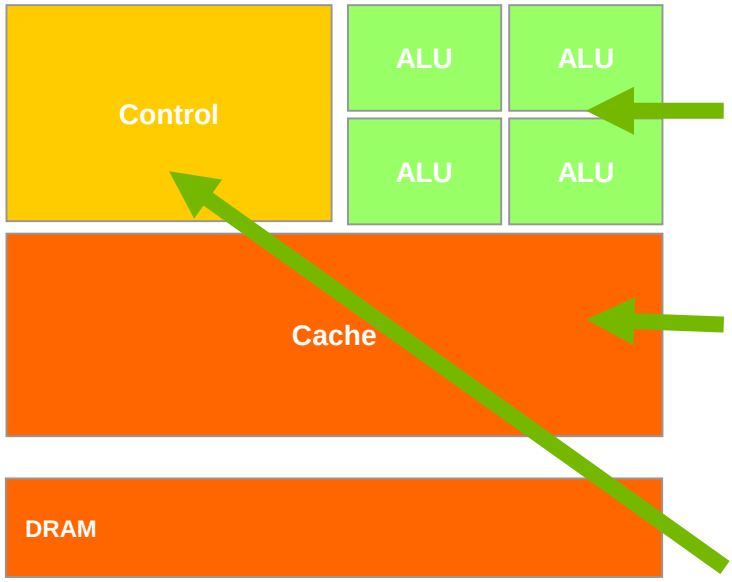
CPU vs GPU



CPU and GPU are designed very differently



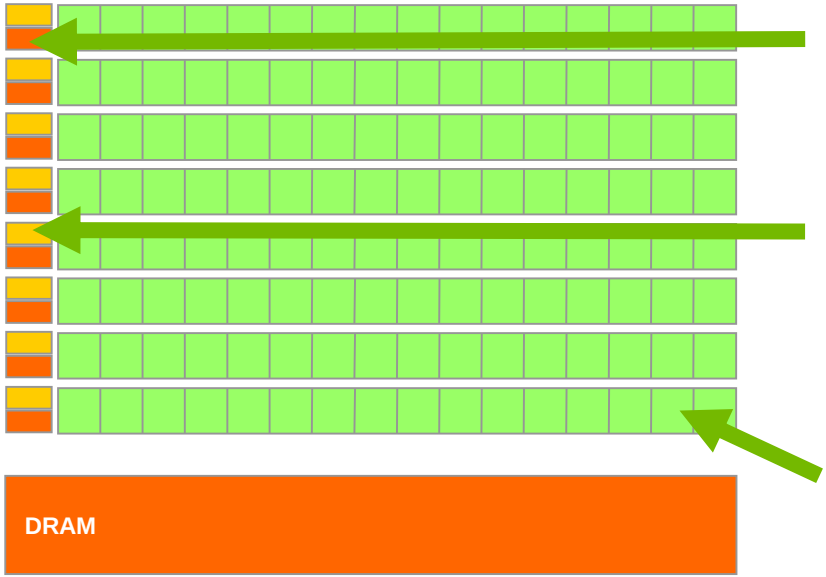
CPUs: Latency Oriented Design



CPU

- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design



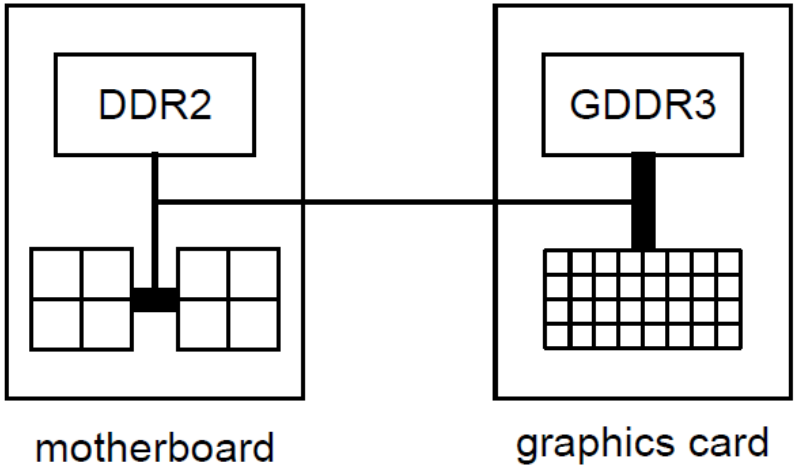
GPU

- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10X+ faster than CPUs for parallel code

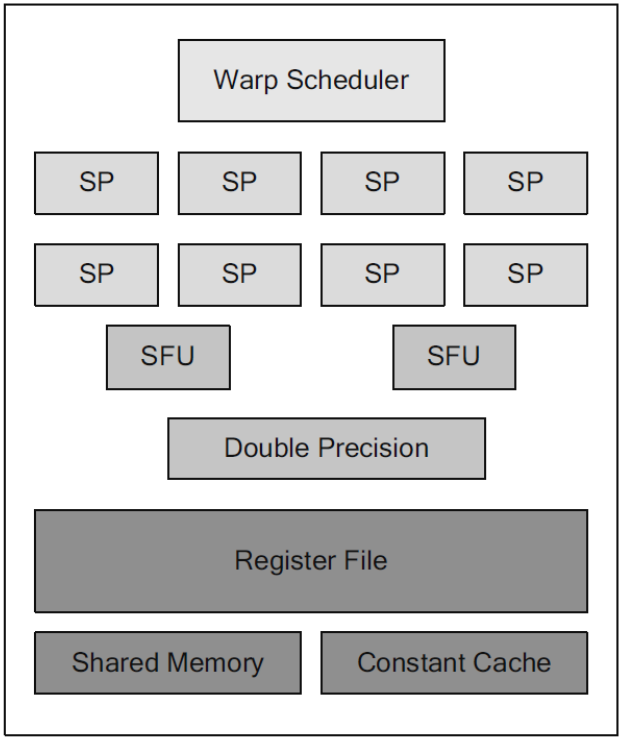
Hardware: CPU – GPU architecture



Hardware: CUDA Compatible GPU

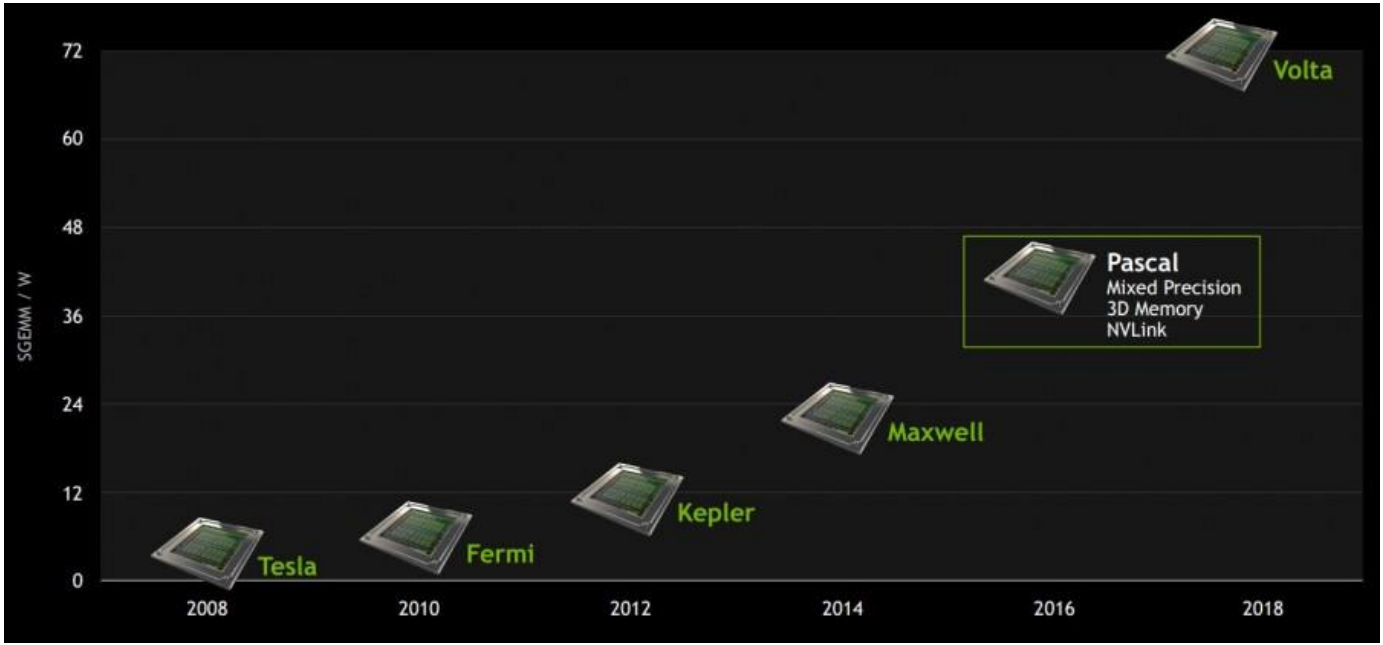
CUDA: Hardware

Streaming Multiprocessors



CUDA: Hardware

CUDA architecture evolution



Hardware view

- 3 generations of hardware cards in use:
- Kepler (compute capability 3.x):
  - first released in 2012
  - includes HPC cards with excellent double precision
- Maxwell (compute capability 5.x):
  - first released in 2014 only gaming cards
  - not HPC, so poor DP
- Pascal (compute capability 6.x):
  - first released in 2016
  - many gaming cards (1050-1080 series) and a few HPC cards (P100)

Hardware view

- Building block is a Streaming Multiprocessor (SM)
  - 128 cores (64 in P100) and 64K registers
  - 96KB (64KB in P100) of shared memory
  - 48KB (24KB in P100) L1 cache
  - 8-16KB cache for constants
  - up to 2K threads per SM
- Different chips have different numbers of these SMs:

Model	SMs	Bandwidth	Memory	Power
GTX 1070	16	256 GB/s	8 GB	120W
GTX 1080	20	320 GB/s	8 GB	150W
GTX Titan X	28	480 GB/s	12 GB	180W

SIMT: Single Instruction Multiple thread

- Key feature of cores within an SM
  - All cores run the same instruction simultaneously but with different data
  - Minimum of 32 threads (warp) performing the same task (almost) at the same time
  - Traditional technique in graphics processing and many scientific applications

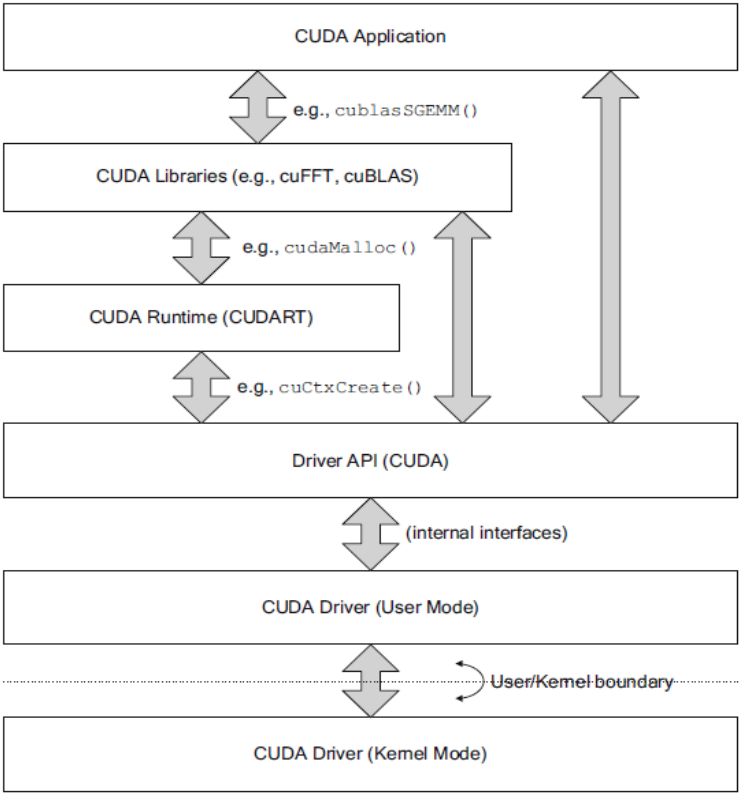
- Spawning large number of threads yield high performance
  - **No penalty for context changes.** Each thread has its own registers which limits the maximum number of active threads.
  - The threads are executed in SM in groups of 32 called "WARPS". The execution alternates between active and temporarily inactive warps.

CUDA: Software

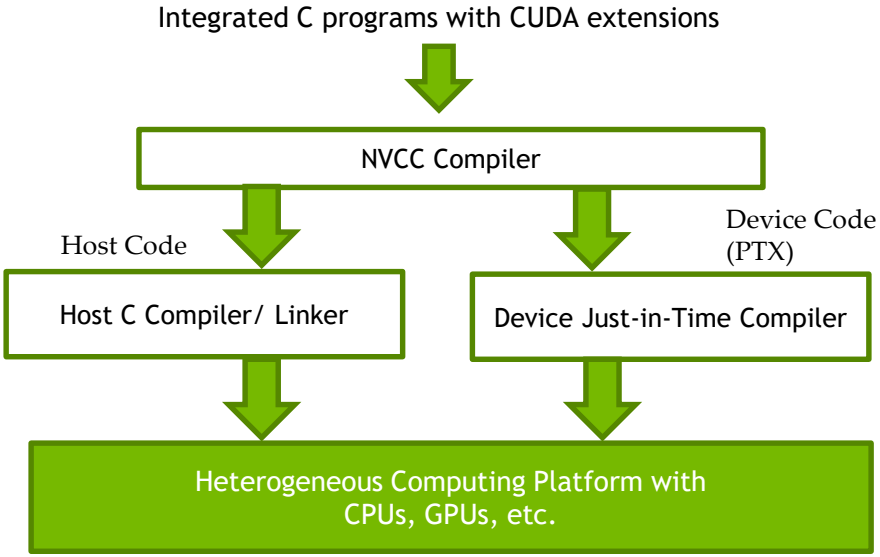
- **CUDA (Compute Unified Device Architecture)**
  - It refers to both a compiler and a set of development tools created by NVIDIA
  - **C-based with some extensions**
  - C++ support, Fortran. Wrappers for other programming languages *.NET, Python, Java, MATLAB, etcetera*
  - **Large number of examples and good documentation,** which reduces the learning curve for those with experience in languages such as OpenMPI and MPI.
  - Extensive user community in **NVIDIA forums**
    - https://devtalk.nvidia.com/

CUDA: Software

CUDA Stack



Compiling A CUDA Program



Software: CUDA program

- A CUDA program has two parts:
  - Host code on the CPU that interfaces with the GPU
  - Kernel code running on the GPU
- At Device level, there are two APIs
  - Runtime: Simpler & easier to use.
  - Driver: more flexible but more complex to use.
    - Driver version doesn't mean more performance, but more flexibility when working with the GPU

Software: CUDA

Basic concepts

– **Kernel**: it is a programming function that will run in parallel on the GPUs.

– **Thread Block**: is a set of threads in 1D, 2D or 3D. Each block is executed on a single SM, but an SM can have several blocks assigned to it for execution.

– **Grid**: is an structure used to group the blocks for execution, either in 1D, 2D or 3D.

Software: CUDA

Basic concepts

Kernel invocation:

```
kernel_routine<<<grid_dim, block_dim>>> (args...);
```

Thread Block and grid size (define using the data type dim3):

```
dim3 block_dim (32, 32, 1); // 1024 threads (32 x 32 x 1) 2D
dim3 grid_dim (4, 4, 1); // 16 thread blocks - grid (4 x 4 x 1) 2D
```

Software: CUDA

Kernel execution and memory addressing

Each thread executes a copy of the kernel, and has the following information:

Variables are passed as parameters to the kernel function (GPU memory pointers)

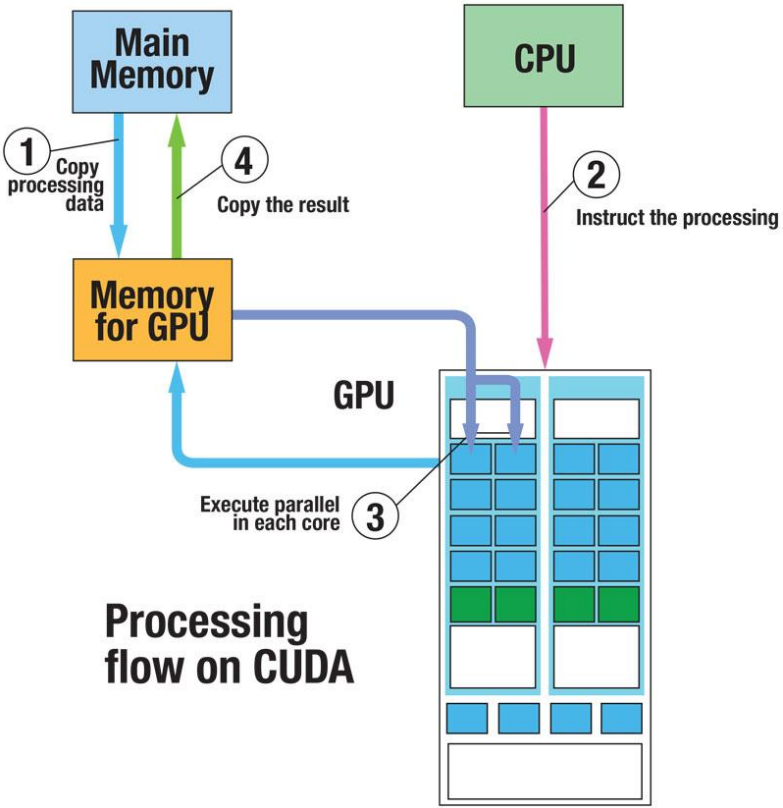
Global constants in GPU memory

Special variables (dim3) for thread identification (built-in):

– **gridDim** (grid size)
– **blockDim** (thread block size)
– **blockIdx** (thread block id)
– **threadIdx** (thread id) (Local numeration for each block)

Software: CUDA

Processing flow on CUDA



Software: CUDA

Compute Capability

Feature support (unlisted features are supported for all compute abilities)	Compute ability (version)									
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.x
Integer atomic functions operating on 32-bit words in global memory	No	Yes								
atomicExch() operating on 32-bit floating point values in global memory										
Integer atomic functions operating on 32-bit words in shared memory	No	Yes								
atomicExch() operating on 32-bit floating point values in shared memory										
Integer atomic functions operating on 64-bit words in global memory										
Warp vote functions										
Double-precision floating-point operations	No			Yes						
Atomic functions operating on 64-bit integer values in shared memory	No				Yes					
Floating-point atomic addition operating on 32-bit words in global and shared memory										
_ballot()										
_threadfence_system()										
_syncthreads_count(), _syncthreads_and(), _syncthreads_or()										
Surface functions										
3D grid of thread block										
Warp shuffle functions	No					Yes				
Funnel shift	No						Yes			
Dynamic parallelism	No							Yes		
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No								Yes	
Atomic addition operating on 64-bit floating point values in global memory and shared memory	No									Yes

Software: CUDA

Device Query

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\1_Utilities\deviceQuery\../../bin/win64/Debug/deviceQuery.exe

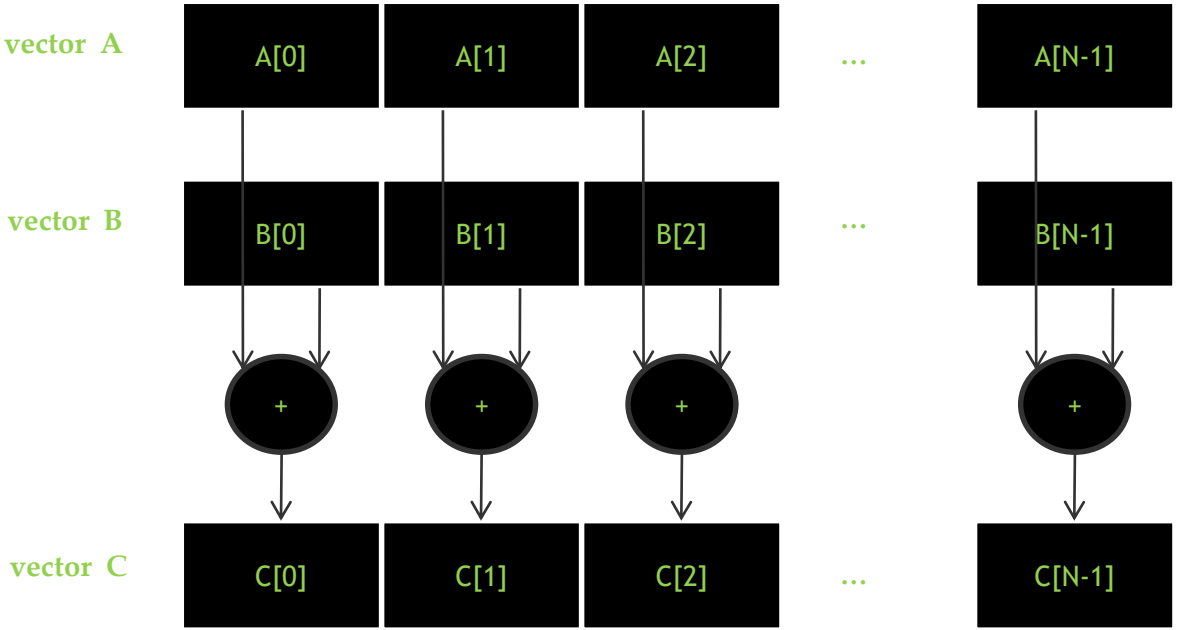
```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\1_Utilities\deviceQuery\../../bin/win64/Debug/deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX TITAN X"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 12288 MBytes (12884901888 bytes)
  (24) Multiprocessors, (128) CUDA Cores/MP:     3072 CUDA Cores
  GPU Max Clock rate:                            1076 MHz (1.08 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 3145728 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
```

Data Parallelism - Vector Addition Example

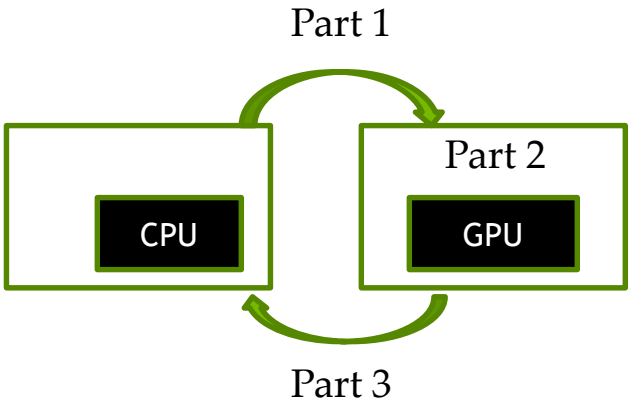


Vector Addition – Traditional C Code

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++)
        h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

Heterogeneous Computing vecAdd CUDA Host Code

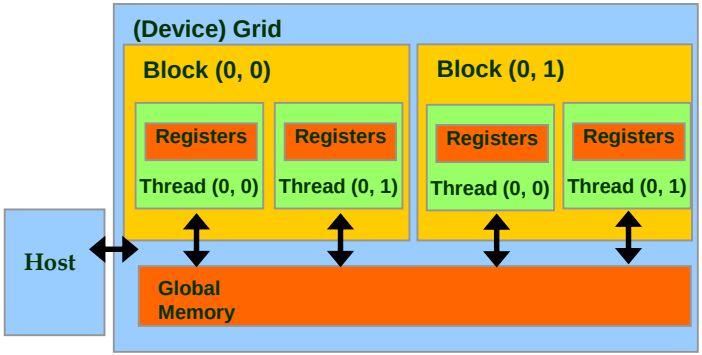



```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code
    // the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

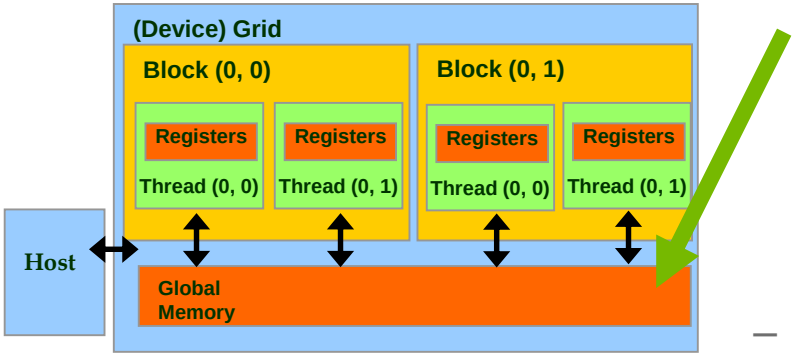
Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

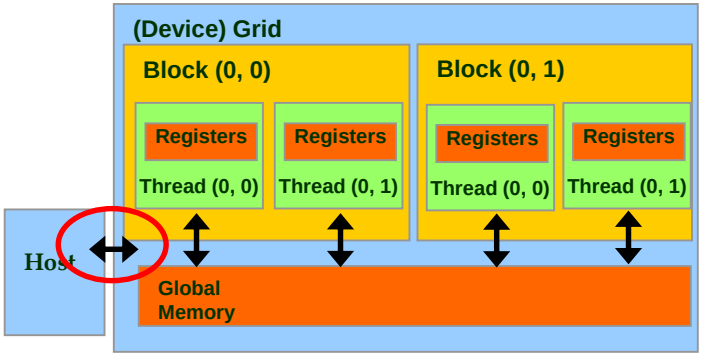
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

Host-Device Data Transfer API functions



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous

Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_B, size);
     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_C, size);

     // Kernel invocation code – to be shown later

     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

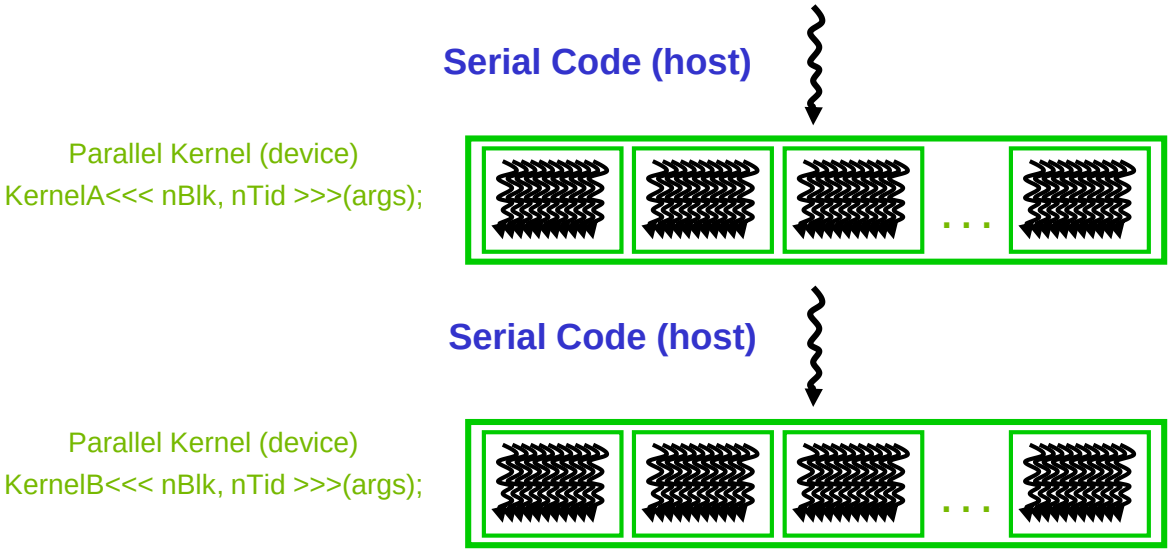
In Practice, Check for API Errors in Host Code

```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

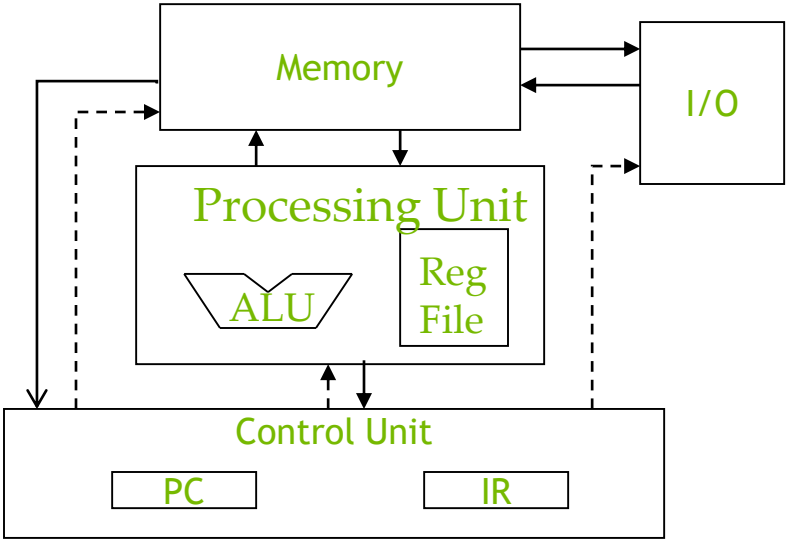
CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code



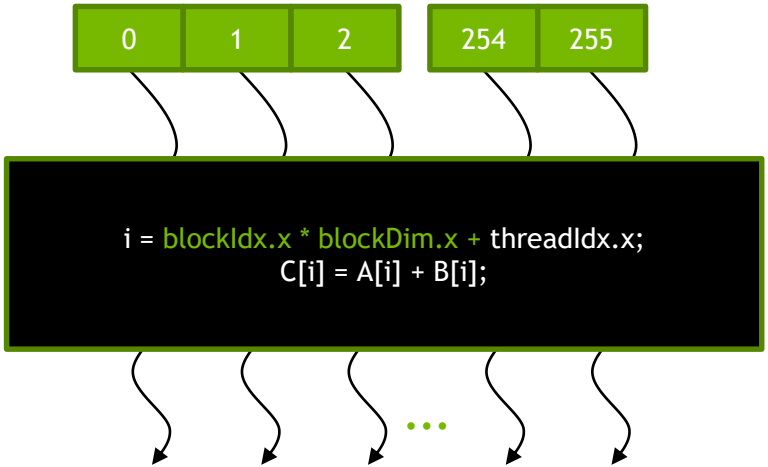
A Thread as a Von-Neumann Processor

A thread is a “virtualized” or “abstracted”
Von-Neumann Processor

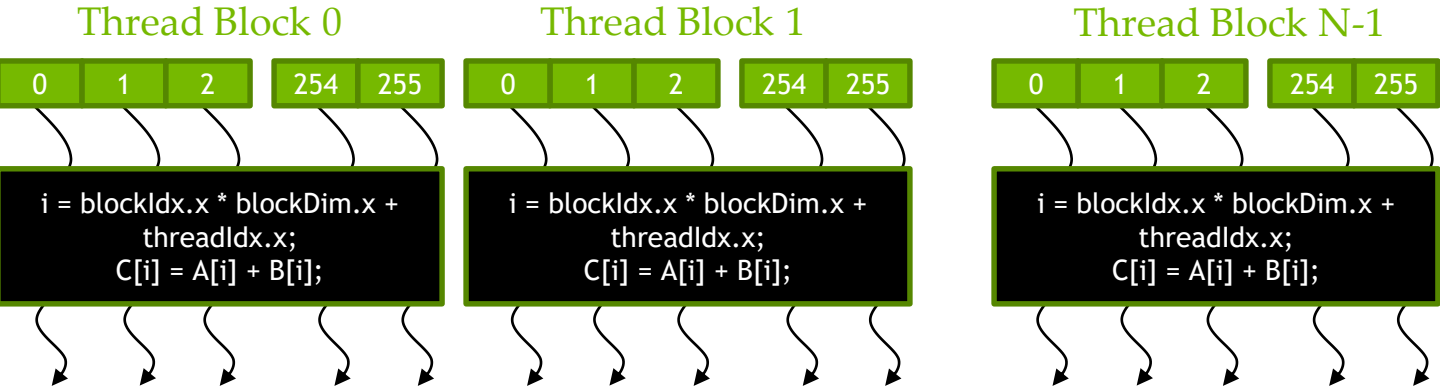


Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



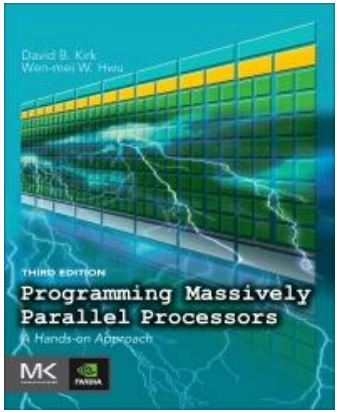
Thread Blocks: Scalable Cooperation

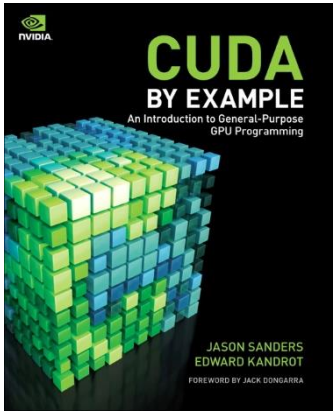


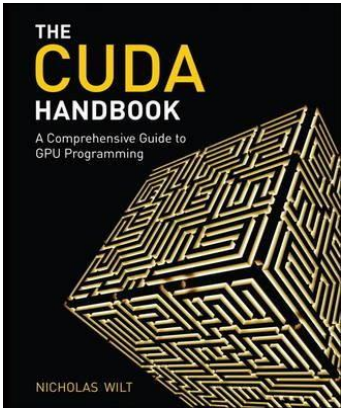
- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

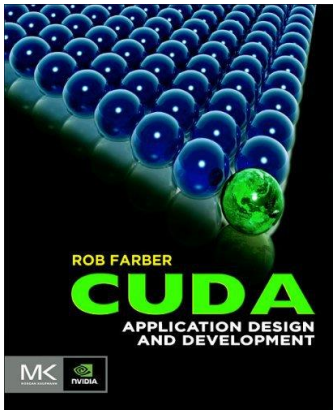
Let´s implement this example using what we we've learnt so far!

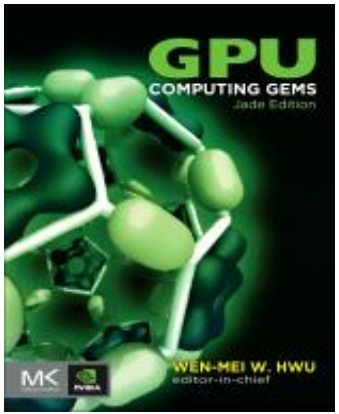
GPU computing reading resources

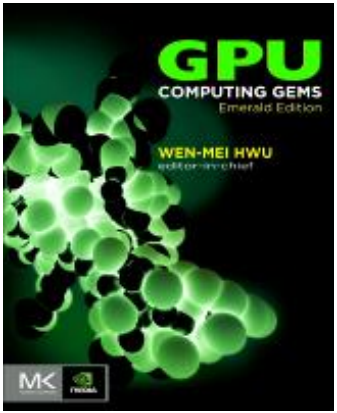












Introduction to parallel programming using CUDA

Hardware & Software

Thanks for your attention!

The slides have been done in collaboration with SERGIO ORTS ESCOLANO!

Albert García García

agarcia @ dtic.ua.es