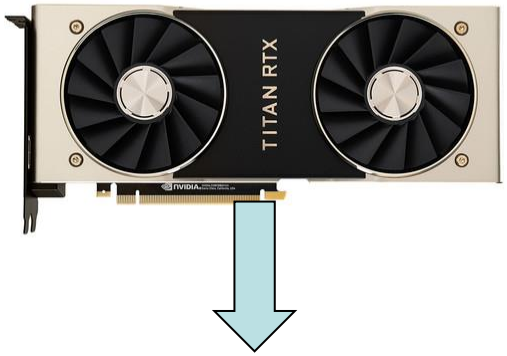
# Introduction to GPU Architecture & Programming

CS140

Tao Yang

# Overview

- Hardware architecture
- Programming model
- Example




# GPU vs CPU

| | CPU | GPU |
|---|---|---|
| Core count | 2–128 (consumer to server models) | Thousands of smaller, simpler cores |
| Clock speed | 3-5Ghz | 1-2Ghz |
| Memory | Cache layers (L1–L3) + system RAM (DDR4/DDR5) | High-bandwidth memory (GDDR6X, HBM2e/HBM3/HBM3e) |

| | Speed | Cost |
|---|---|---|
| CPU (AMD Ryzen 9. Intel Core Ultra 9) | 1-3 TFLOPS<br>60 TFLOPS with SIMD | ~$400 |
| Consumer GPUs (NVIDIA RTX 5090) | 3 - 800 TFLOPS | $600-$3000 |
| Datacenter AI GPUs (AMD Instinct MI. NVIDIA H100/Blackwell) | 100-2000 TFLOPS | $25,000-$70,000 |

# GPU Programming API

- **CUDA (Compute Unified Device Architecture) : parallel GPU programming API created by NVIDA**
  - Hardware and software architecture for issuing and managing computations on GPU
  - Massively parallel architecture with **lots of fine-grain** thread-level parallelism
  - API libraries with C/C++/Fortran language
    - **Numerical libraries: cuBLAS, cuFFT,**
- **OpenGL** – an open standard for GPU programming
- **DirectX** – Microsoft multimedia programming interfaces
- **Hot technology but challenging to learn**:
  - NVIDA-specific teminology. Not consistent with others. Software portability issues.
  - More hardware constraints affecting software design.

# GPU Architecture

**SP**: scalar processor
'CUDA core'

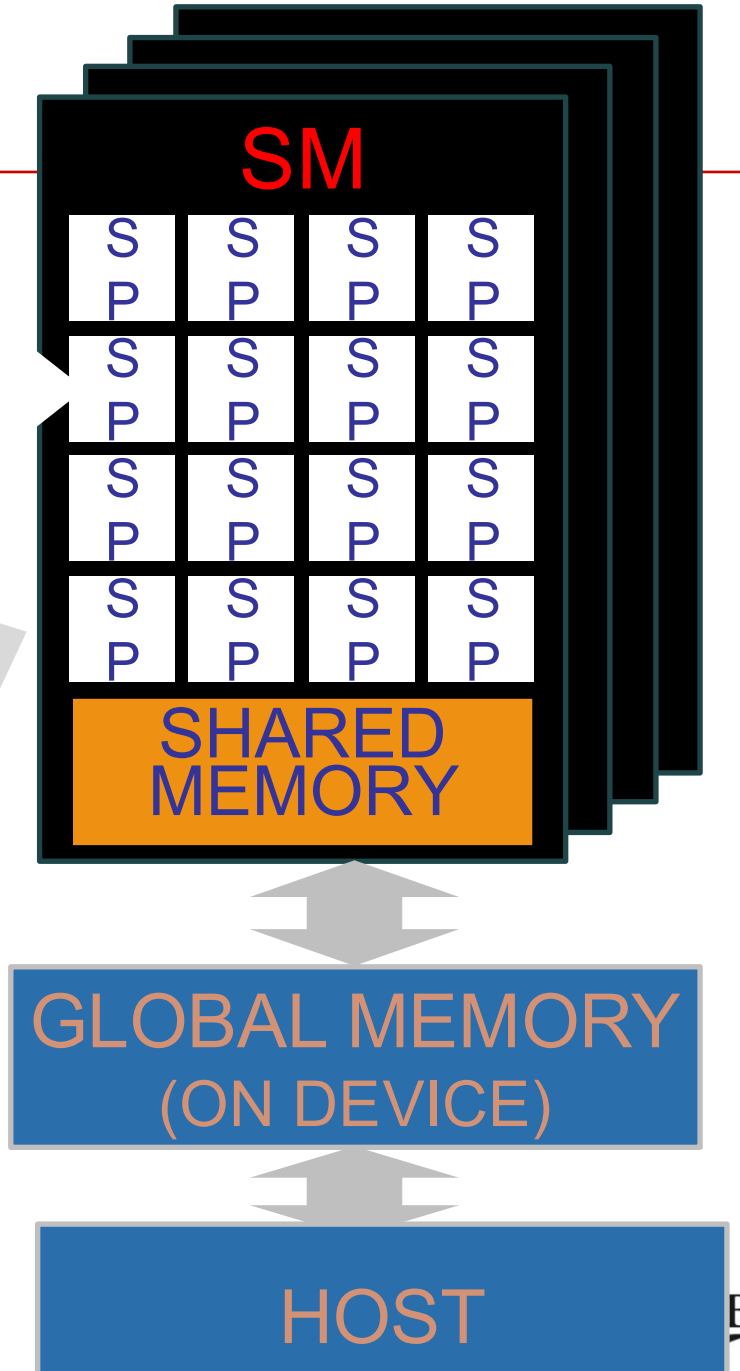
Executes one thread

**SM**

**streaming multiprocessor**

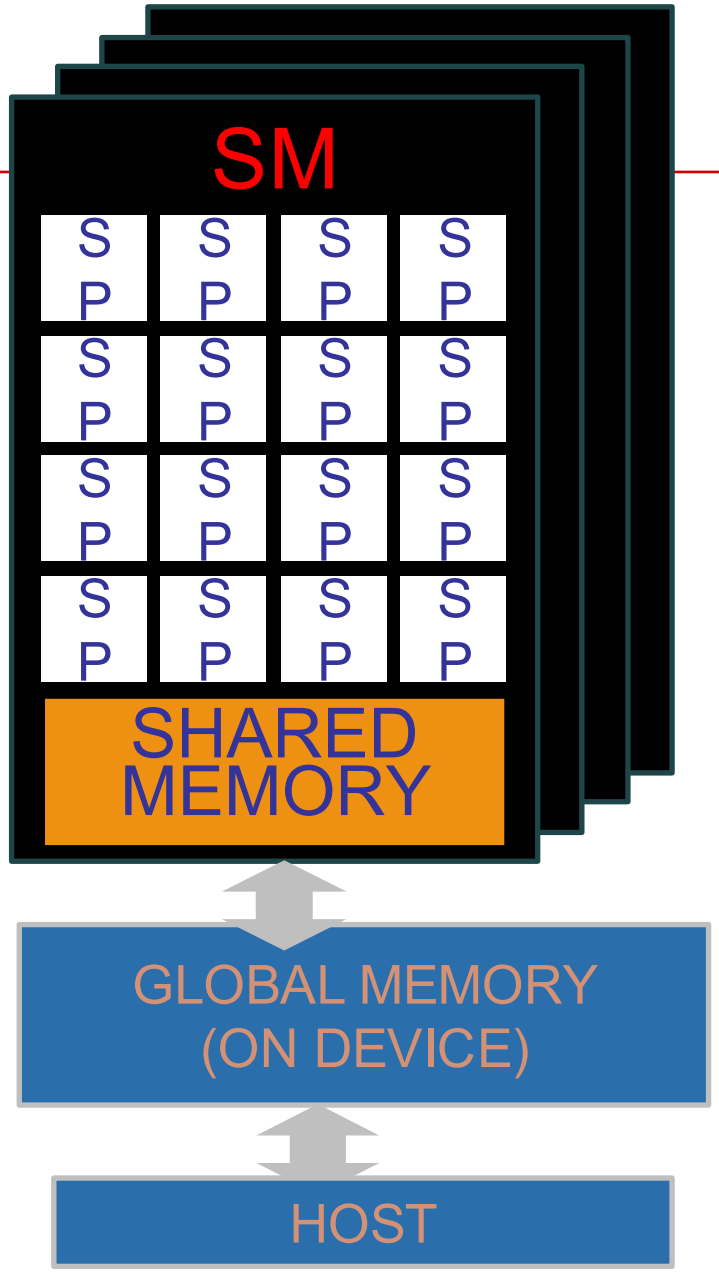
**32xSP (or 16, 48 or more)**

**Fast local 'shared memory'**
**(shared between SPs)**
**16 KiB (or 64 KiB)**



# GPU on Expanse

- **Each node has four V100 GPUs**
- **Each GPU has 80 SM units.**
  - Each SM unit has 64Cuda cores. Default 96KB shared memory
  - Total 5120 Cuda cores.
    - 15 teraflops for single precision
    - 7.8 teraflops for double precision
  - 32GB global memory with 900GB/s memory bandwidth
- **Intel 6248 Xeon:** 40 cores with 2.5GHz. 364GB memory



# CUDA Essentials

- **Visit developer.nvidia.com. Download**
  - Driver
  - Toolkit (compiler nvcc)
  - SDK (examples) (recommended)

**Book/refrences:**

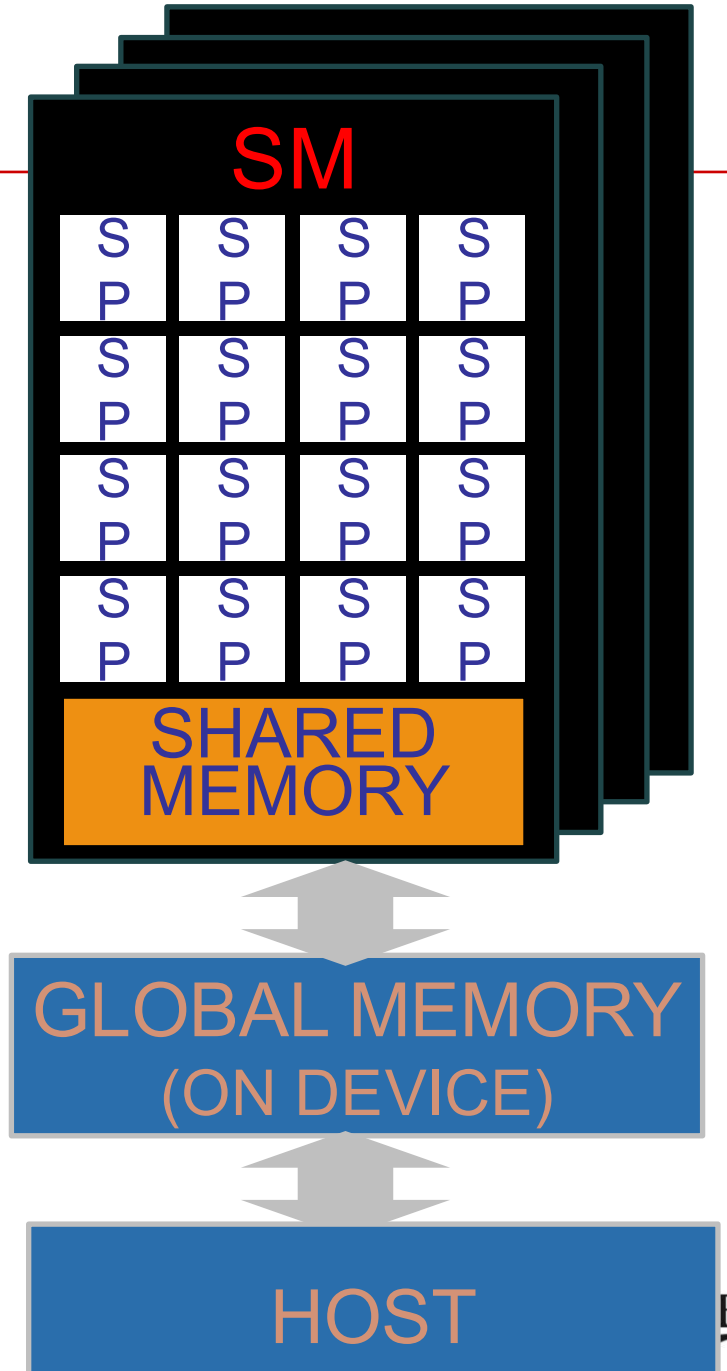
•CUDA Programmers guide by NVIDA.
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index
•CUDA by Example. Book by Sander/Kandrot. NVIDA. 2011

**Other tools:**

- 'Emulator'. Executes on CPU. Slow
- Simple profiler
- cuda-gdb (Linux)

# How to Program GPUs

- Decompose a program to threads
- **Memory**
  - Shared memory only within each streaming multiprocessor
  - Global memory, accessible by all cores
- **Thread communication**
  - Communicate through shared or global memory
  - Barrier available within single SM
    - Not easy to synchronize threads across SMs.

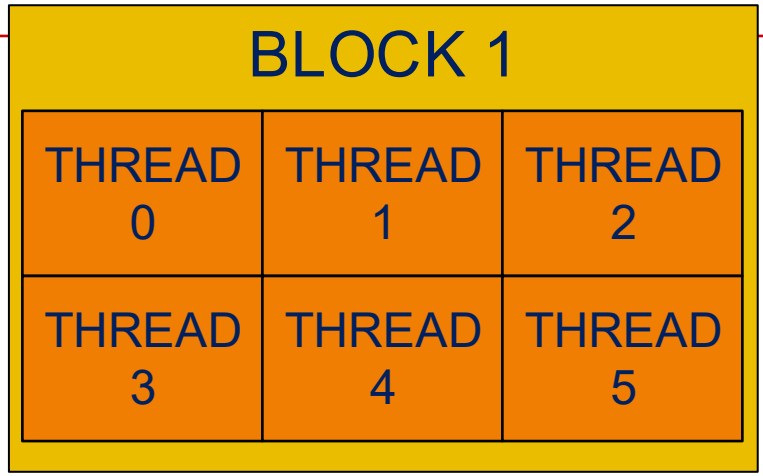


# GPU Threads vs CPU Threads

- **Differences between GPU and CPU threads**
  - GPU threads are extremely lightweight
    - Very little creation overhead in GPU vs. much more in CPU
  - GPU needs thousands of threads for full efficiency
    - Multi-core CPU needs a few
- **The GPU is viewed as a compute device that:**
  - Is a coprocessor to the CPU host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the GPU device as kernels which run in parallel on many threads

# Organization of Threads: Thread Blocks

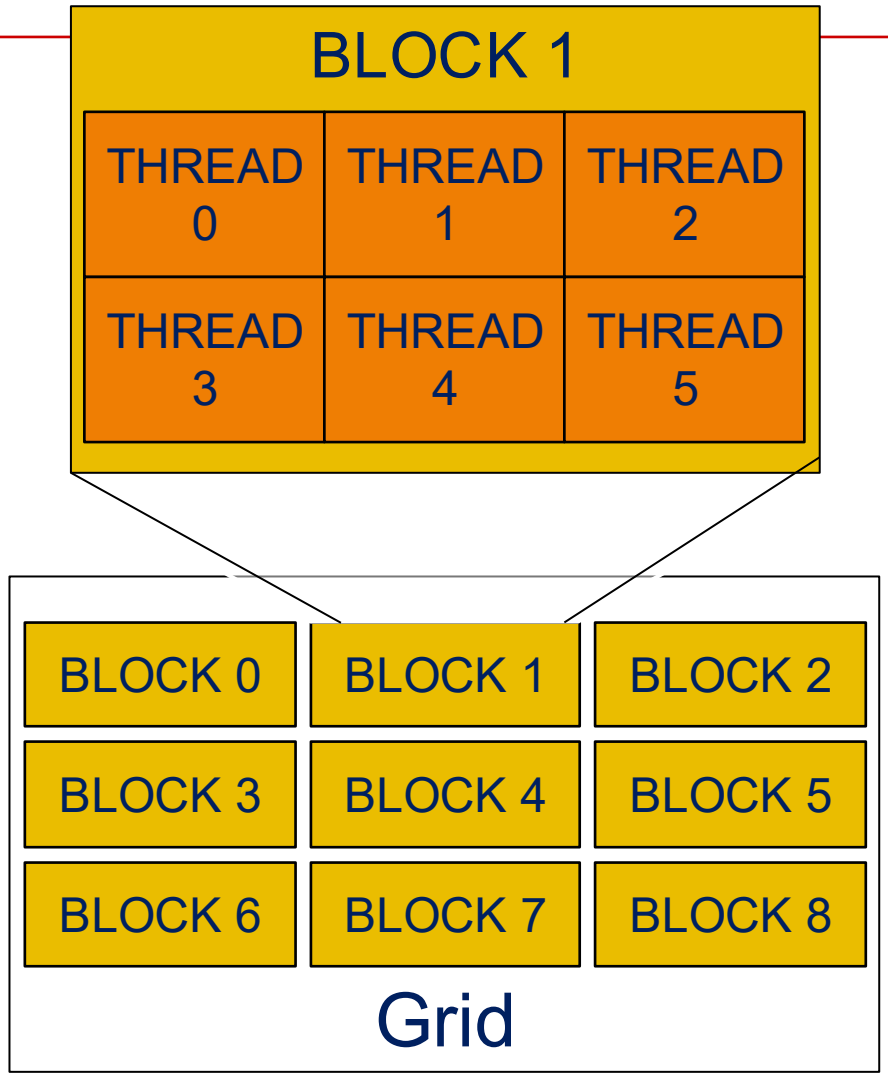
- **Threads grouped in thread blocks**
  - There is the maximum size limit per block
  - 512 or 1024




- Computation executed is one thread block, and then another thread block
- One thread block executes within one SM
  - All threads sharing the 'shared memory' of one SM
  - 32 threads are executed simultaneously by OS (called 'warp')
- Different blocks can run on different SMs
  - execute in parallel and independently!

# Cooperation in Thread Blocks

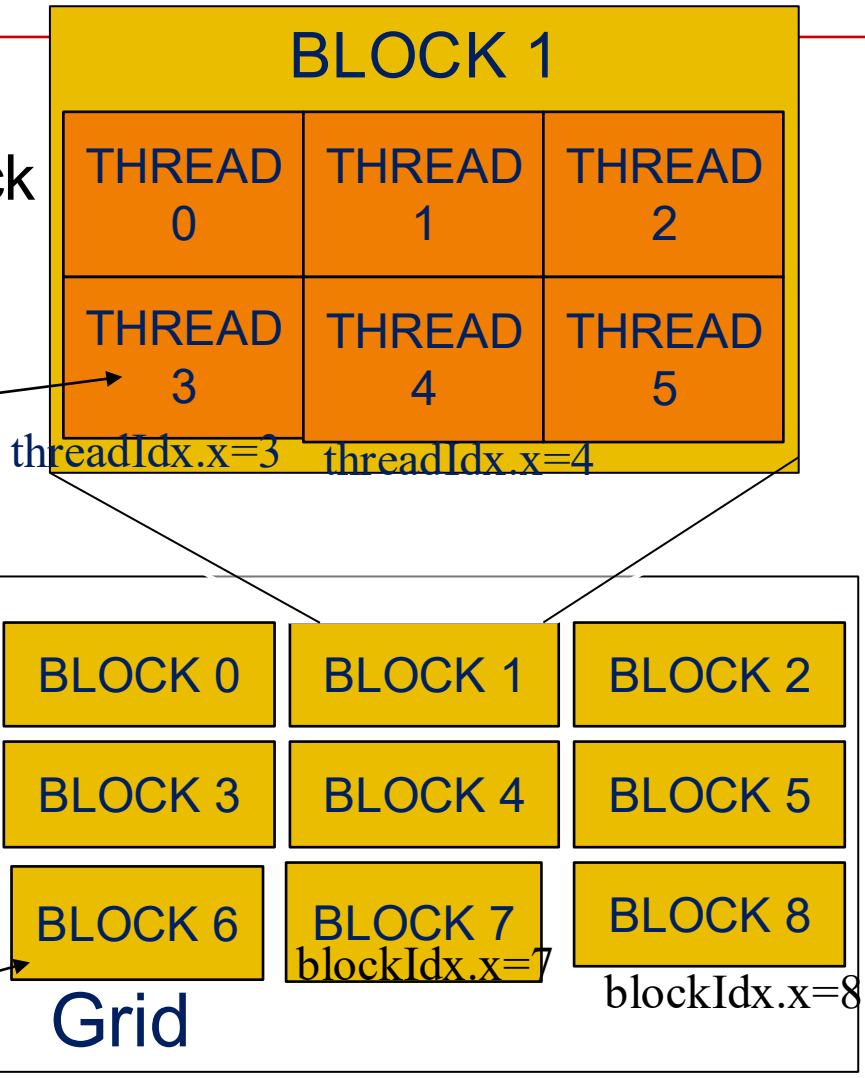
- **A thread block is a batch of threads that can cooperate with each other by:**
  - Synchronizing their execution
    - Use __syncthreads()
  - Efficiently sharing data through a low latency shared memory
- **Two threads from two different blocks cannot cooperate**
  - Use cudaDeviceSynchronize() for all threads



# Thread Grouping: Grids and Blocks

- Blocks of threads form a **GRID**
- **Thread ID:** unique within a block

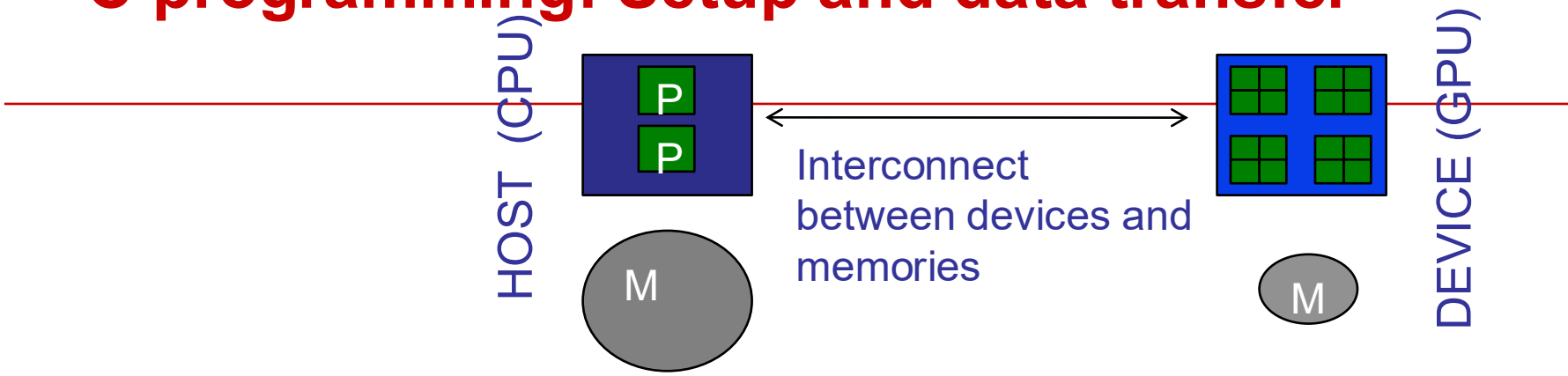
For 1D numbering of threads within a block, use special variable **threadIdx.x**
Size of a block is in variable **blockDim.x** (e.g. 6)




- **Block ID:** unique within a grid

For 1D numbering of blocks, use special variable **blockIdx.x**

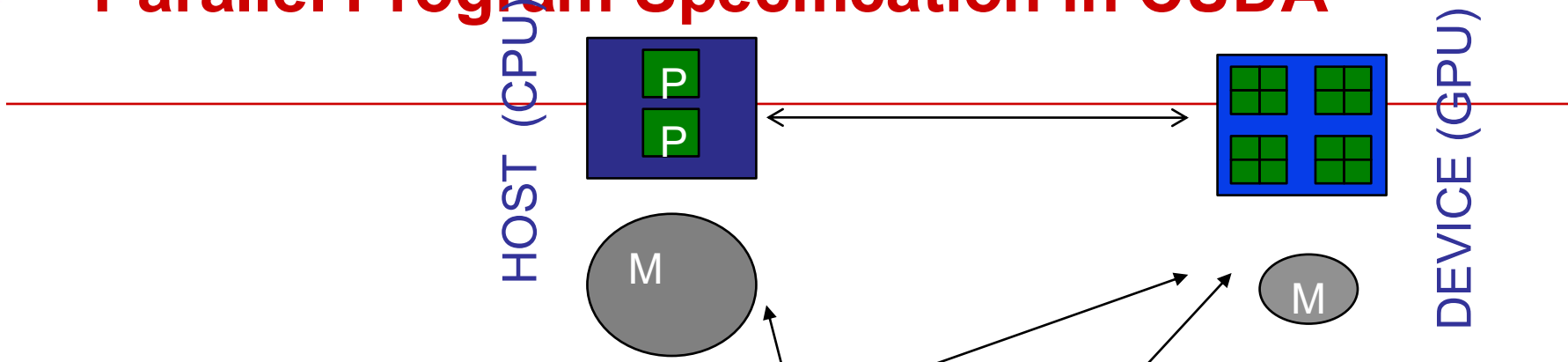
# C programming: Setup and data transfer




- **GPU is the 'device', CPU is the 'host'**
- **`cudaMalloc`**
  - Allocate global memory on GPU
- **`cudaMemcpy`**
  - transfer data to and from global memory of GPU

```
cudaMemcpy(d_in_array, h_in_array,
        SIZE*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(h_out_array,d_out_array,
        BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
```

# Parallel Program Specification in CUDA




- **Computation partitioning (where to run: host vs. GPU)**
  - Declarations on functions __host__, __global__, __device__
  - Specification of thread programs that run on device:
    - kernelFunc <<<noBlocks, noThreads>>>( arguments).
    - This program is called ``kernel''.
- **Data partitioning on GPU (where does data reside, who may access it and how?)**
  - Declarations on data __shared__, __device__, __constant__, …
- **Concurrency management of threads within an SM**
  - __synchthreads()

# Code that executes on GPU: Kernels

- **Kernel**
  - A C function which return `void`
  - Executes on GPU **in parallel** on the specified blocks of threads
  - The keyword `__global__` tells the compiler nvcc to make a function a kernel (and compile/run it for the GPU, instead of the CPU)
  - Called from the host side using
    - kernelFunc <<<noBlocks, noThreads>>>(arguments);
    - e.g. kernelFunc<<1,256>>>();
      - Run this kernel function on 1 block of 256 threads with no argument
- **Device functions** can only be called from other device or global functions. __device__ functions cannot be called from host code

# Summary on CUDA Function Declarations

| | Executed on the: | Only callable from the: |
|---|---|---|
| `__device__ float DeviceFunc()` | device | device |
| `__global__ void KernelFunc()` | device | host |
| `__host__ float HostFunc()` | host | host |

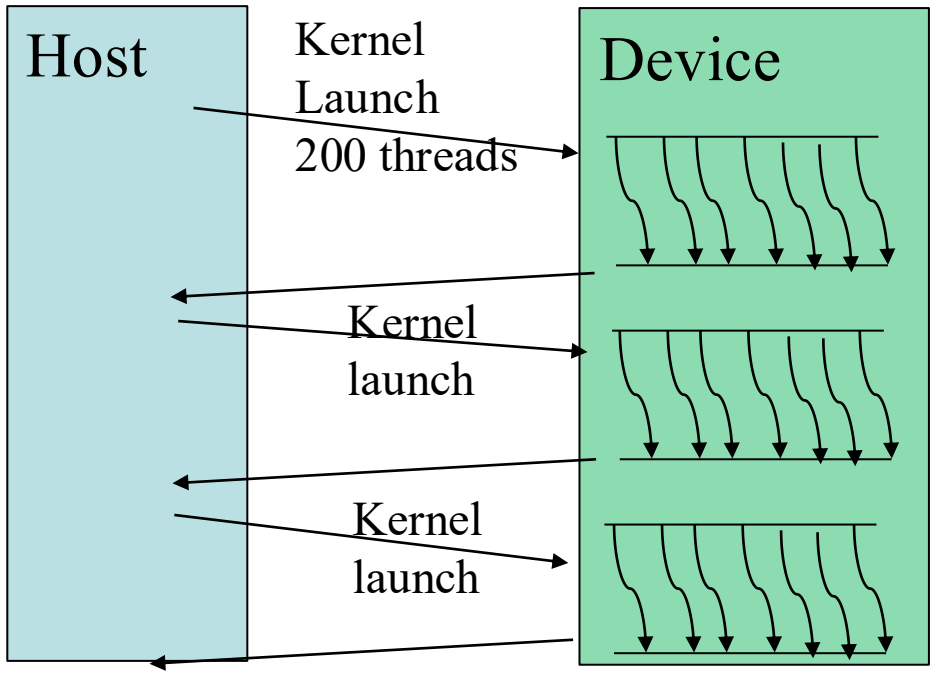
- A kernel function must return `void`

# Fork-join Parallelism with Kernel Calls

Use **consecutive kernel calls** from CPU host to fork threads and implicitly join these threads.

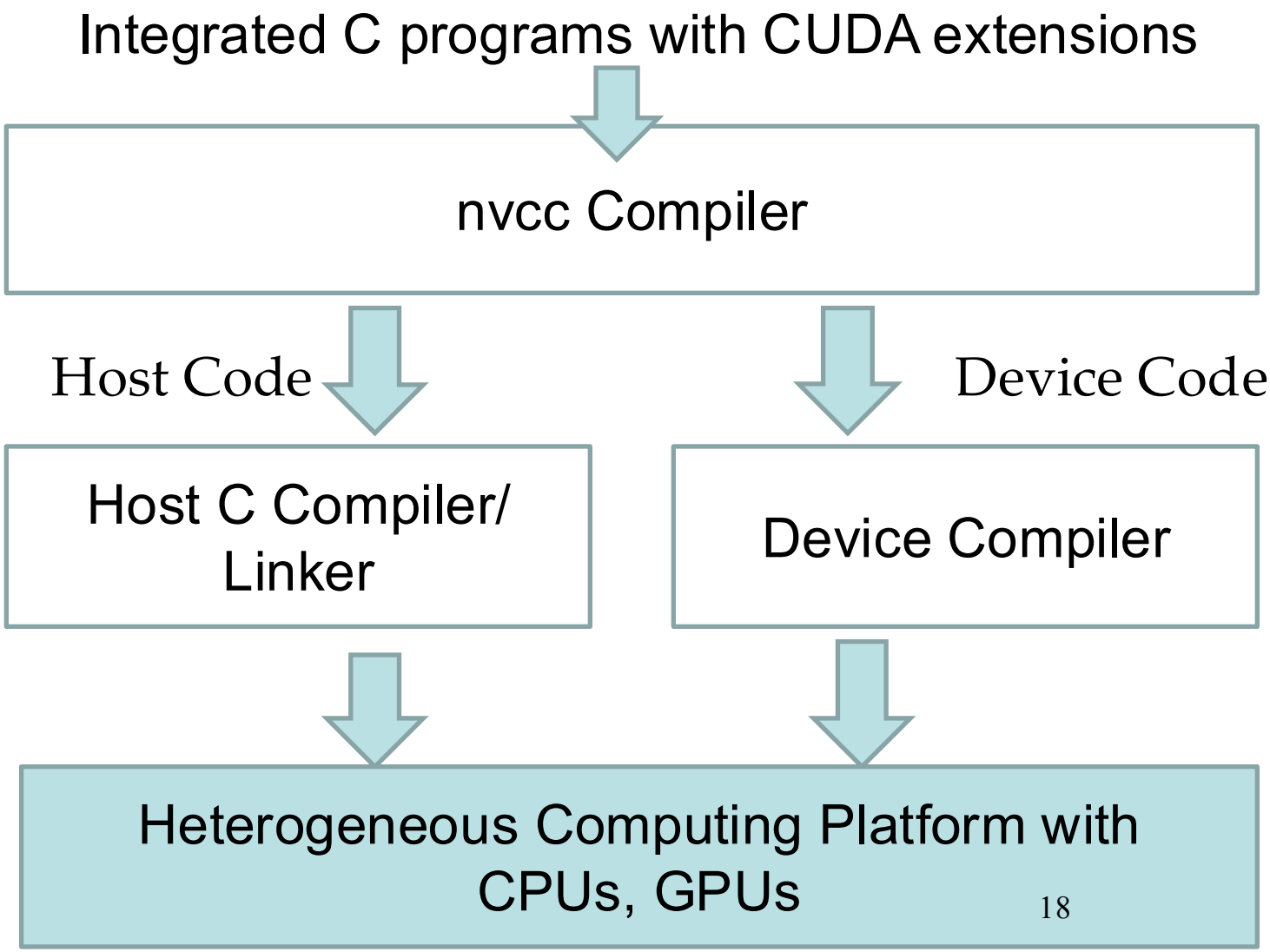
**Host code to run a kernel on 2 blocks of threads and each block has 100 threads**:

```
for(i=0;i<3;i++){
  kernelFunc<<<2, 100>>>();
}
```

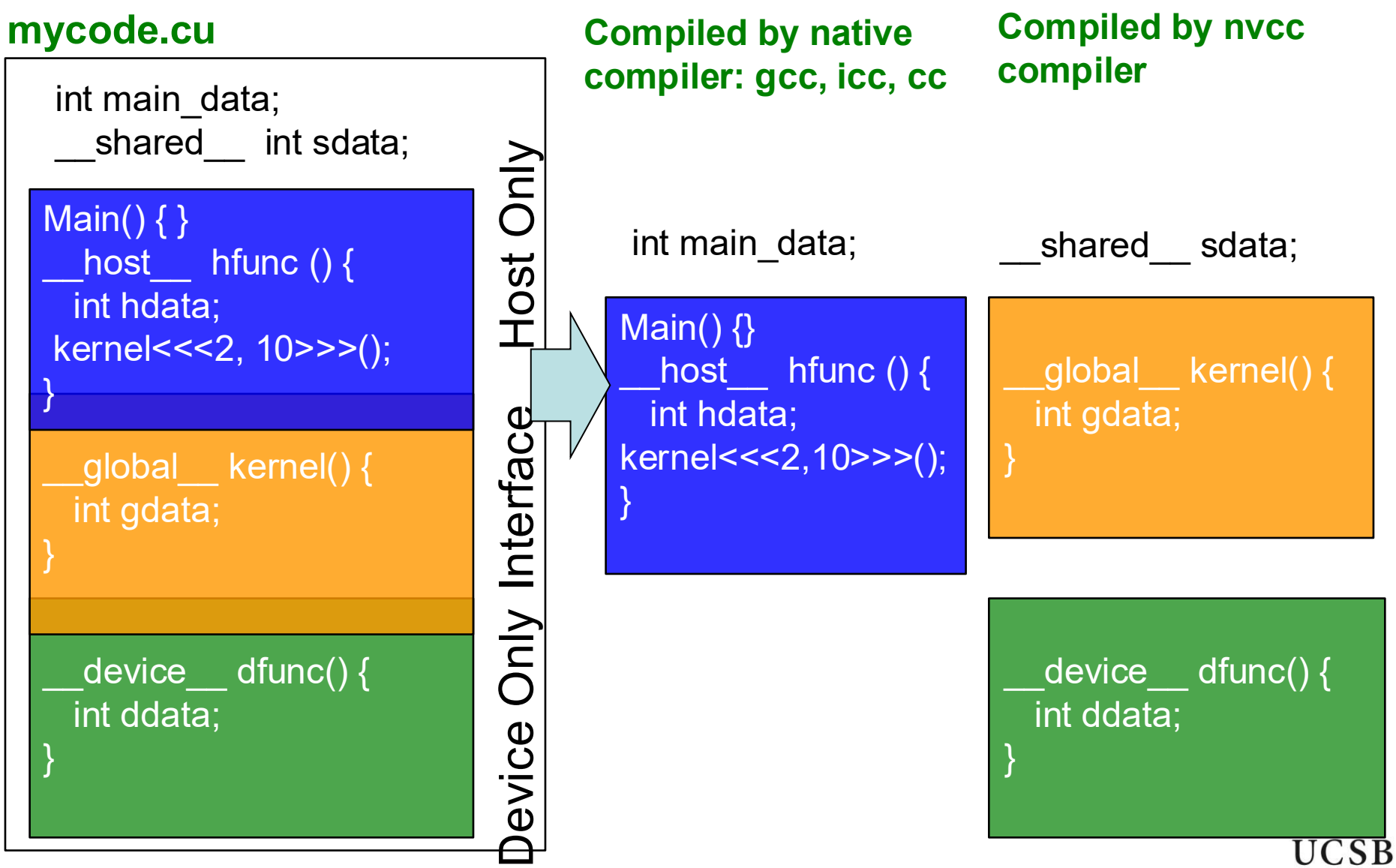



Latest CUDA version allows kernel streams. Use cudaDeviceSynchronize() to join threads

# Compiling A CUDA Program

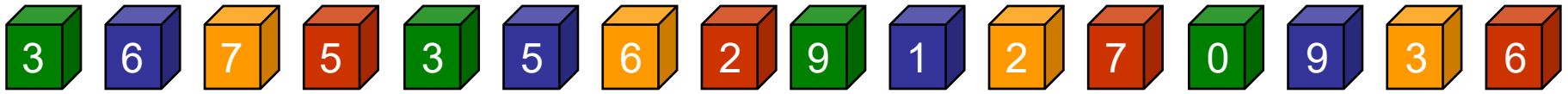


# NVCC Compiler's Role: Partition Code and Compile for Device



# Simple working code example

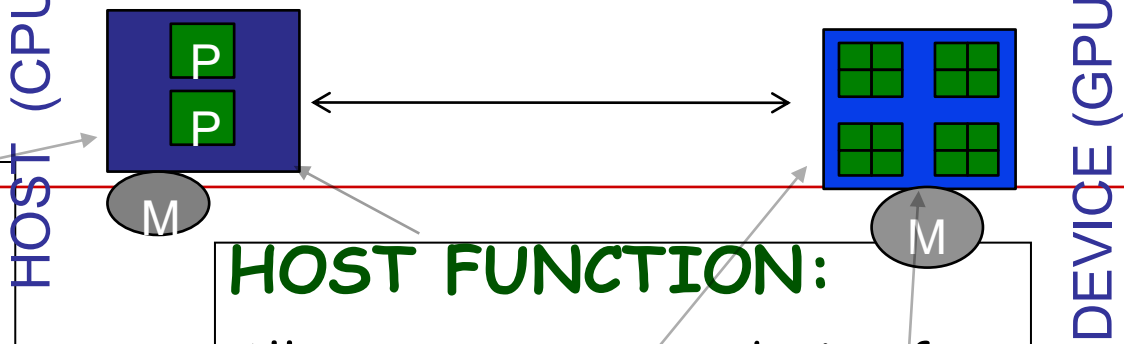
- **What does it do?**
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid




threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Known as a cyclic data distribution

# CUDA Pseudo-Code




## MAIN PROGRAM:

**Initialization**

- **Allocate memory on host for input and output**
- **Assign random numbers to input array**

**Call *host* function**

**Calculate final output from per-thread output**

**Print result**

## HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** kernel function

Synchronize after completion

Copy ***device*** output to host

## GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with "6"

Compute local result

## DEVICE FUNCTION:

Compare current element and "6"

Return 1 if same, else 0

UCSB

# Main Program: Preliminaries

**MAIN PROGRAM:**

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
  int *in_array, *out_array;
  …
}
```

# Main Program: Invoke Global Function

## MAIN PROGRAM:

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute (int
  *in_arr, int *out_arr);

int main(int argc, char **argv)
{
  int *in_array, *out_array;
  /* initialization */ …
 outer_compute(in_array, out_array);
   …
}
```

# Main Program: Calculate Output & Print Result

## MAIN PROGRAM:

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

**Calculate final output from per-thread output**

**Print result**

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute (int
  *in_arr, int *out_arr);

int main(int argc, char **argv)
{
  int *in_array, *out_array;
  int sum = 0;
  /* initialization */ …
  outer_compute(in_array, out_array);
  for (int i=0; i<BLOCKSIZE; i++) {
    sum+=out_array[i];
  }
  printf ("Result = %d\n",sum);
}
```

# Host Function: Preliminaries & Allocation

## HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Synchronize after completion

Copy ***device*** output to host

```
__host__ void outer_compute (int
  *h_in_array, int *h_out_array) {

  int *d_in_array, *d_out_array;

  cudaMalloc((void **) &d_in_array,
        SIZE*sizeof(int));

  cudaMalloc((void **) &d_out_array,
        BLOCKSIZE*sizeof(int));

  ...
}
```

# Host Function: Copy Data To/From Host

## HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Synchronize after completion

Copy ***device*** output to host

```
__host__ void outer_compute (int
  *h_in_array, int *h_out_array) {
  int *d_in_array, *d_out_array;

  cudaMalloc((void **) &d_in_array,
        SIZE*sizeof(int));

  cudaMalloc((void **) &d_out_array,
        BLOCKSIZE*sizeof(int));

  cudaMemcpy(d_in_array, h_in_array,
        SIZE*sizeof(int),
        cudaMemcpyHostToDevice);

  … do computation …

  cudaMemcpy(h_out_array,d_out_array,
        BLOCKSIZE*sizeof(int),
        cudaMemcpyDeviceToHost);

}
```

# Host Function: Setup & Call Global Kernel Function

**HOST FUNCTION:**

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Synchronize after completion

Copy ***device*** output to host

```
__host__ void outer_compute (int
  *h_in_array, int *h_out_array) {
  int *d_in_array, *d_out_array;

  cudaMalloc((void **) &d_in_array,
        SIZE*sizeof(int));

  cudaMalloc((void **) &d_out_array,
        BLOCKSIZE*sizeof(int));

  cudaMemcpy(d_in_array, h_in_array,
        SIZE*sizeof(int),
        cudaMemcpyHostToDevice);

compute<<<1,BLOCKSIZE>>> (d_in_array,
  d_out_array);
cudaThreadSynchronize();

  cudaMemcpy(h_out_array, d_out_array,
        BLOCKSIZE*sizeof(int),
        cudaMemcpyDeviceToHost);

}
```

# Global Function: How to distribute tasks?

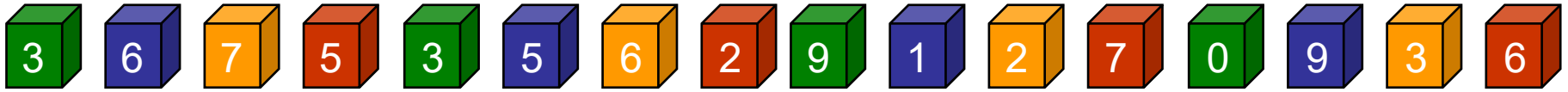
## GLOBAL FUNCTION:

Thread scans subset of array elements

Call *device* function to compare with "6"

Compute local result

```
__global__ void compute(int *d_in,int *d_out) {
  d_out[threadIdx.x] = 0;
  for (int i=0; i<SIZE/BLOCKSIZE; i++) {
    int val = d_in[i*BLOCKSIZE + threadIdx.x];
    d_out[threadIdx.x] += compare(val, 6);
  }
}
```




threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Cyclic distribution

# Device Function

**DEVICE FUNCTION:**

Compare current element and "6"

Return 1 if same, else 0

```
__device__ int
compare(int a, int b) {
  if (a == b) return 1;
  return 0;
}
```

# Summary

- **Introduction to CUDA: C API supporting heterogeneous data-parallel CPU+GPU execution**
  - Computation partitioning
  - Data partitioning (parts of this implied by decomposition into threads)
  - Data organization and management
  - Concurrency management
- **Compiler nvcc takes as input a .cu program and produces**
  - C Code for host processor (CPU), compiled by native C compiler
  - Code for device processor (GPU), compiled by nvcc compiler