Distributed Systems

Subheadline

- Stefan Henkler **E-Mail:** stefan.henkler@hshl.de

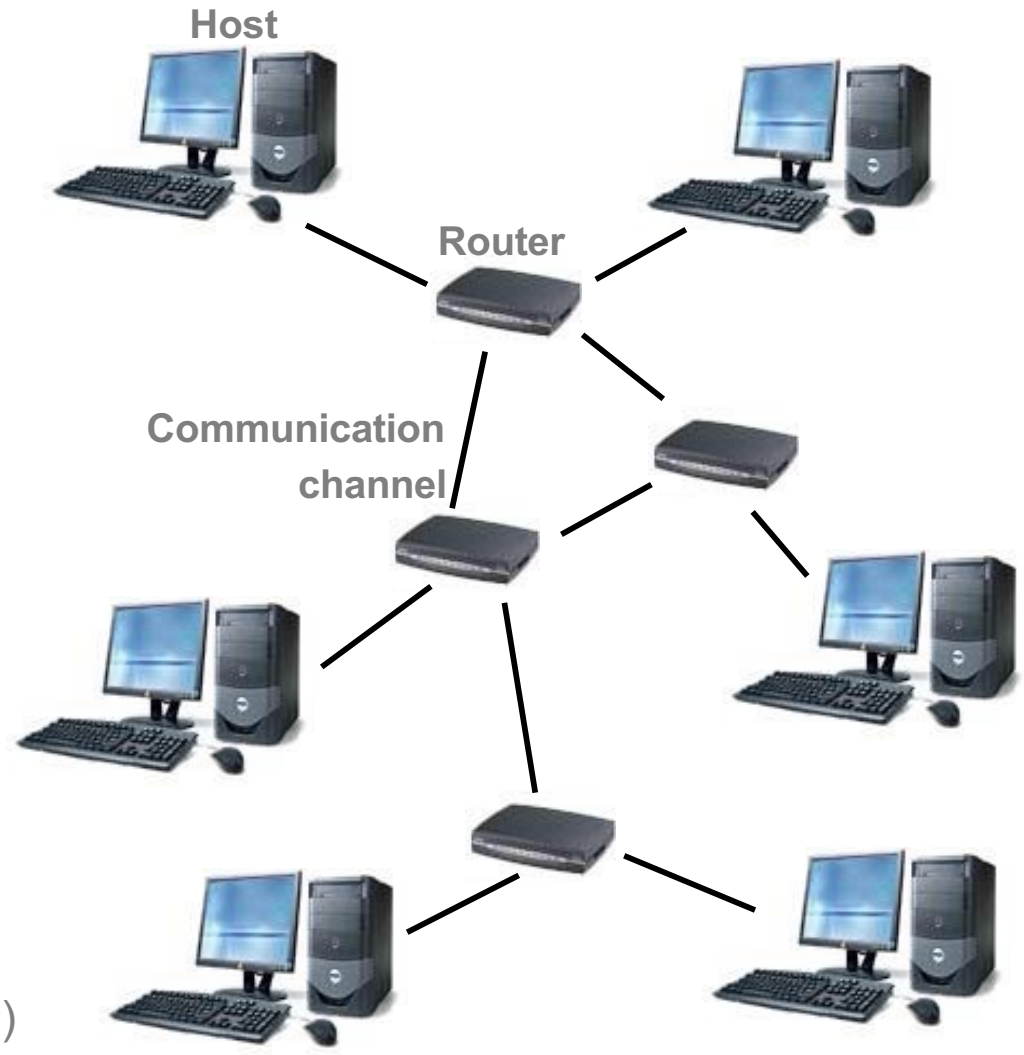
Overview

I. Introduction - parallel architectures
II. Distributed Architectures
III. GPU architecture and programming

Recap distributed architectures and deepen socket programming

Introduction

- Computer Network
  - hosts, routers, communication channels
- **Hosts** run applications
- **Routers** forward information
- **Packets**: sequence of bytes
  - contain control information
  - e.g. destination host
- **Protocol** is an agreement
  - meaning of packets
  - structure and size of packets
  - e.g. Hypertext Transfer Protocol (HTTP)



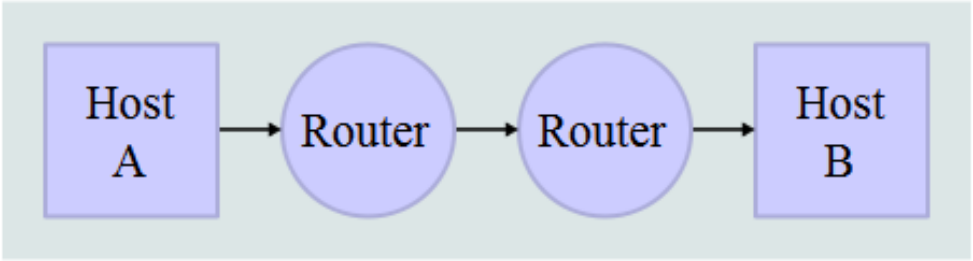
Protocol Families - TCP/IP

- Several protocols for different problems
  - e.g. TCP/IP

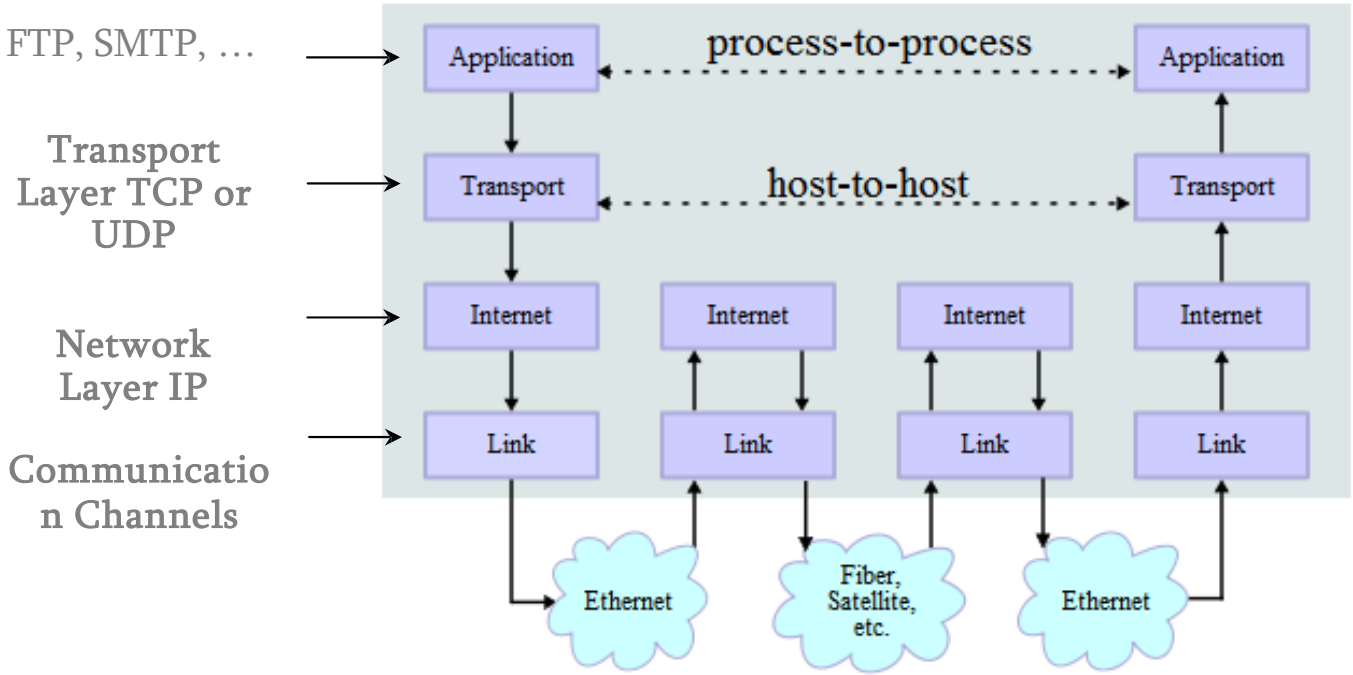
- TCP/IP provides end-to-end connectivity specifying how data should be
  - formatted,
  - addressed,
  - transmitted,
  - routed, and
  - received at the destination
- can be used in the internet and in stand-alone private networks
- it is organized into layers

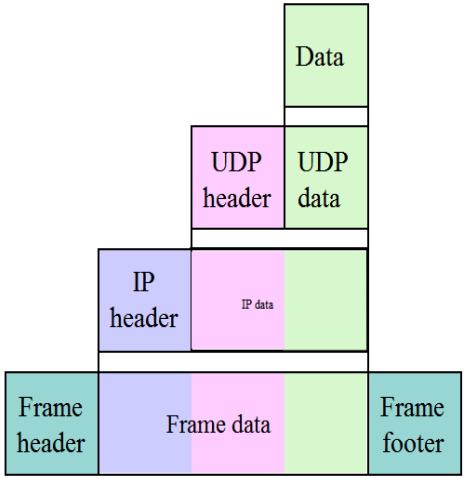
TCP/IP

Network Topology



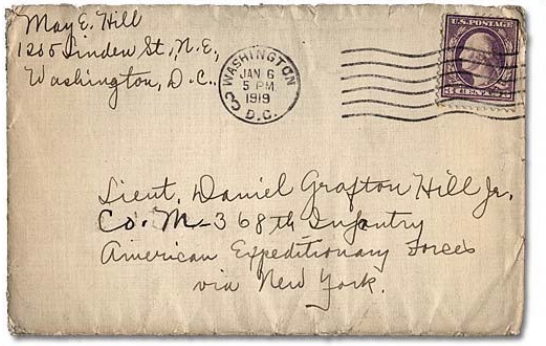
Data Flow





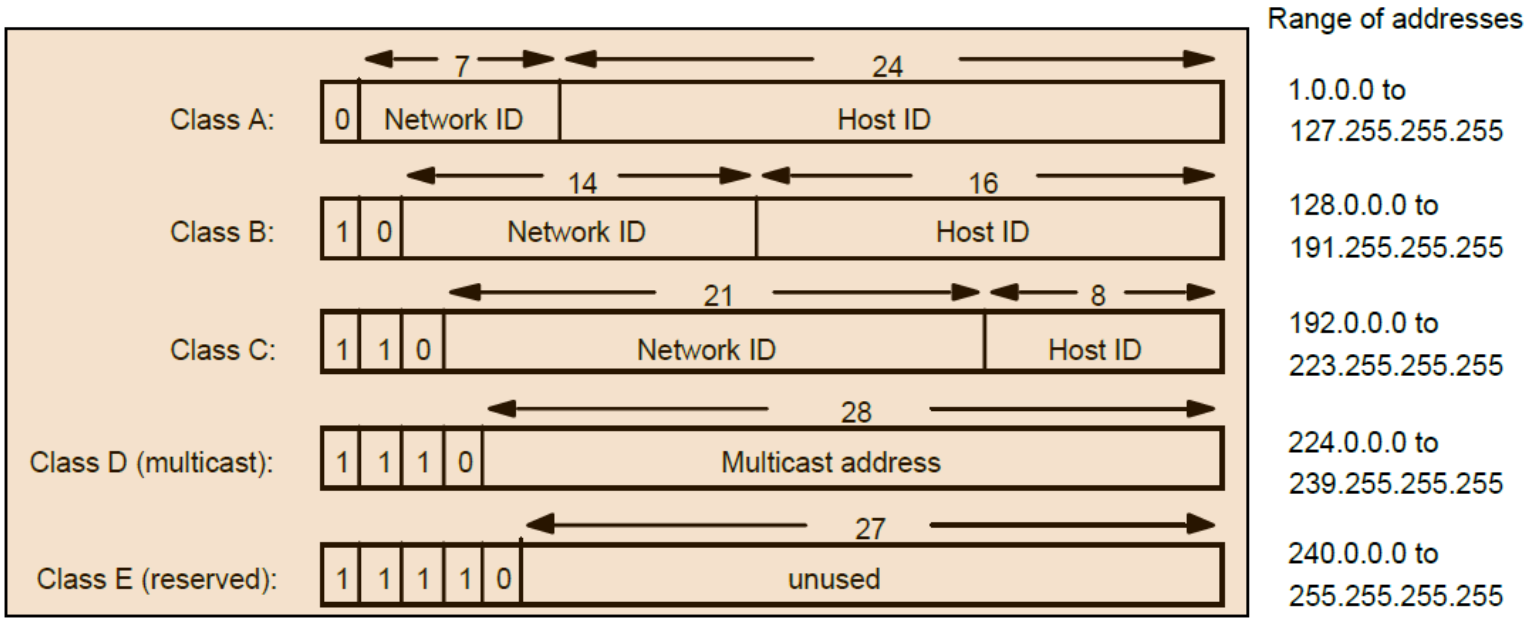
Internet Protocol (IP)

- provides a **datagram** service
  - packets are handled and delivered independently
- **best-effort** protocol
  - may loose, reorder or duplicate packets
- each packet must contain an **IP address** of its destination



Addresses

- The **32** bits of an IPv4 address are broken into **4 octets**, or 8 bit fields (0-255 value in decimal notation).
- For networks of different size,
  - the first one for large networks to three for small networks
    - octets can be used to identify the **network**, while
  - the rest of the octets can be used to identify the **node** on the network.



Range of addresses

1.0.0.0 to
127.255.255.255

128.0.0.0 to
191.255.255.255

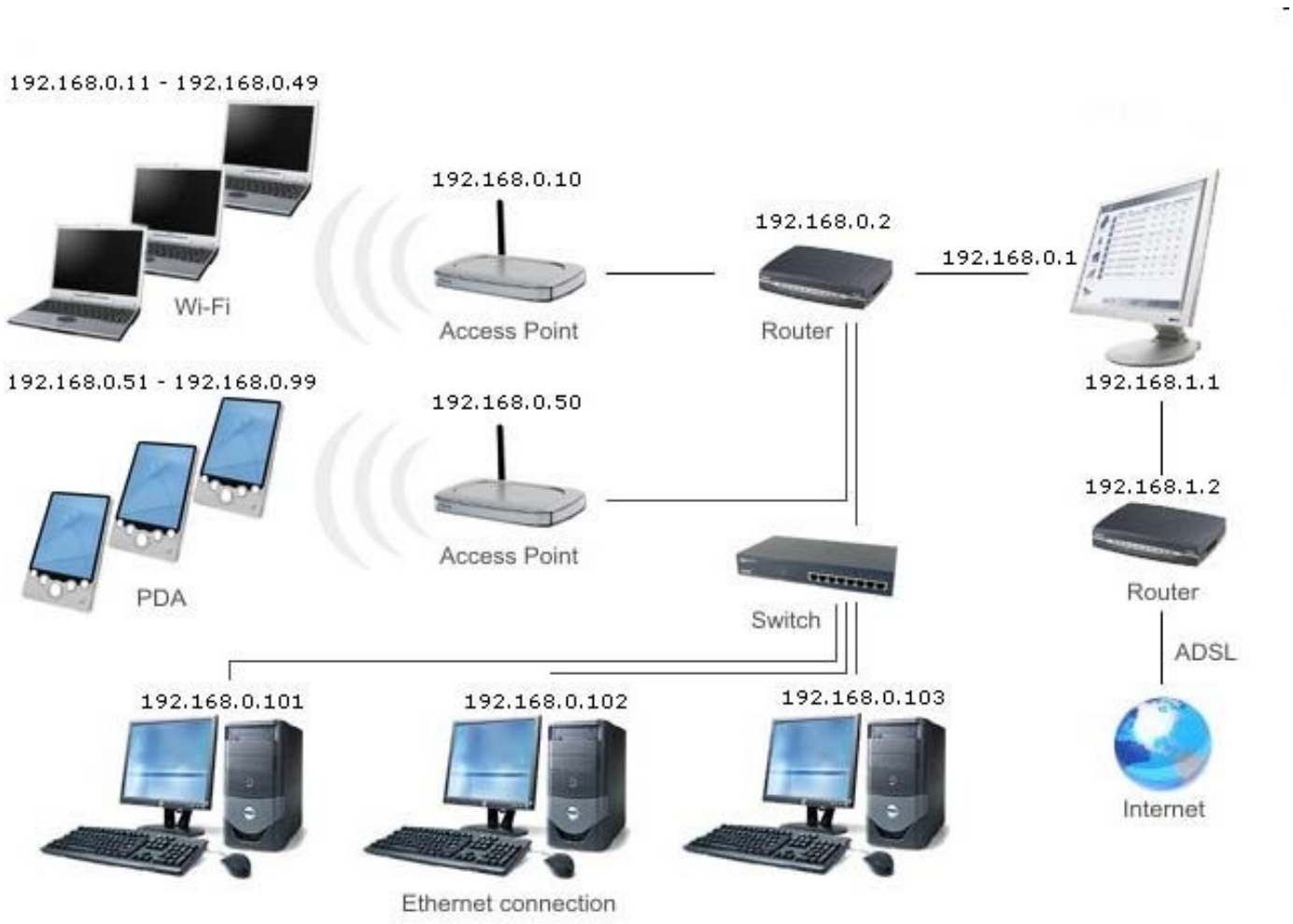
192.0.0.0 to
223.255.255.255

224.0.0.0 to
239.255.255.255

240.0.0.0 to
255.255.255.255

  - Classless Inter-Domain Routing (CIDR)

Local Area Network Addresses - IPv4



TCP vs UDP

- Both use **port numbers**
  - application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to 65535
  - …to provide **end-to-end** transport
- UDP: User Datagram Protocol
  - no acknowledgements
  - no retransmissions
  - out of order, duplicates possible
  - connectionless, i.e., app indicates destination for each packet
- TCP: Transmission Control Protocol
  - reliable **byte-stream channel** (in order, all arrive, no duplicates)
    - similar to file I/O
  - flow control
  - connection-oriented
  - bidirectional

TCP vs UDP

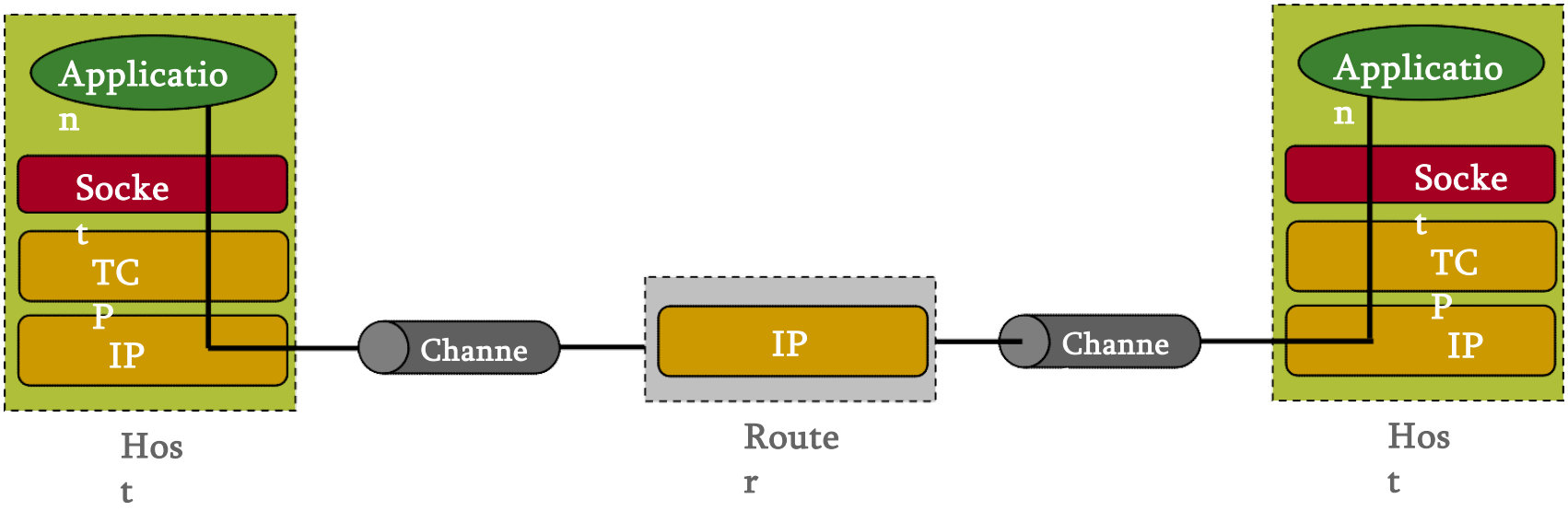
- TCP is used for services with a large data capacity, and a persistent connection
- UDP is more commonly used for quick lookups, and single use query-reply actions.

- Some common examples of TCP and UDP with their default ports:

DNS lookup	UDP	53
FTP	TCP	21
HTTP	TCP	80
POP3	TCP	110
Telnet	TCP	23

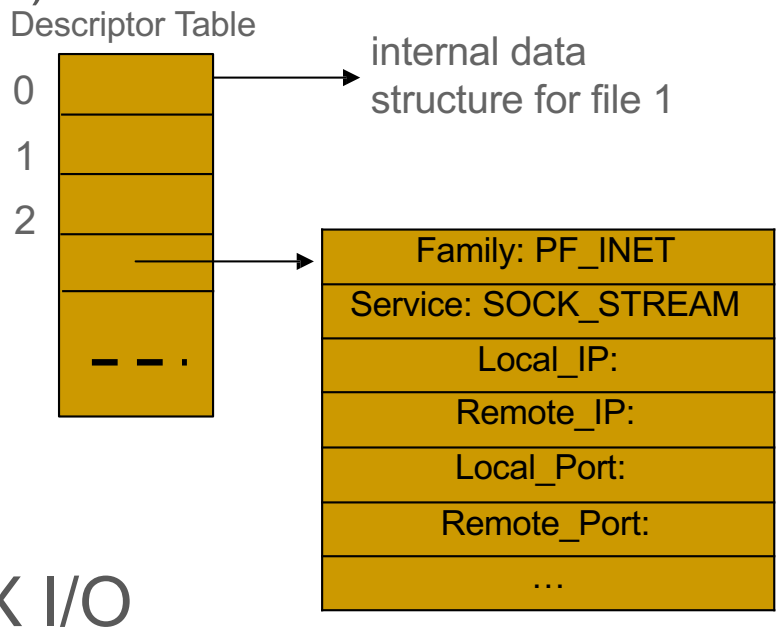
Berkley Sockets

- Universally known as **Sockets**
- It is an abstraction through which an application may send and receive data
- Provide **generic access** to interprocess communication services
  - e.g. IPX/SPX, Appletalk, TCP/IP
- Standard API for networking

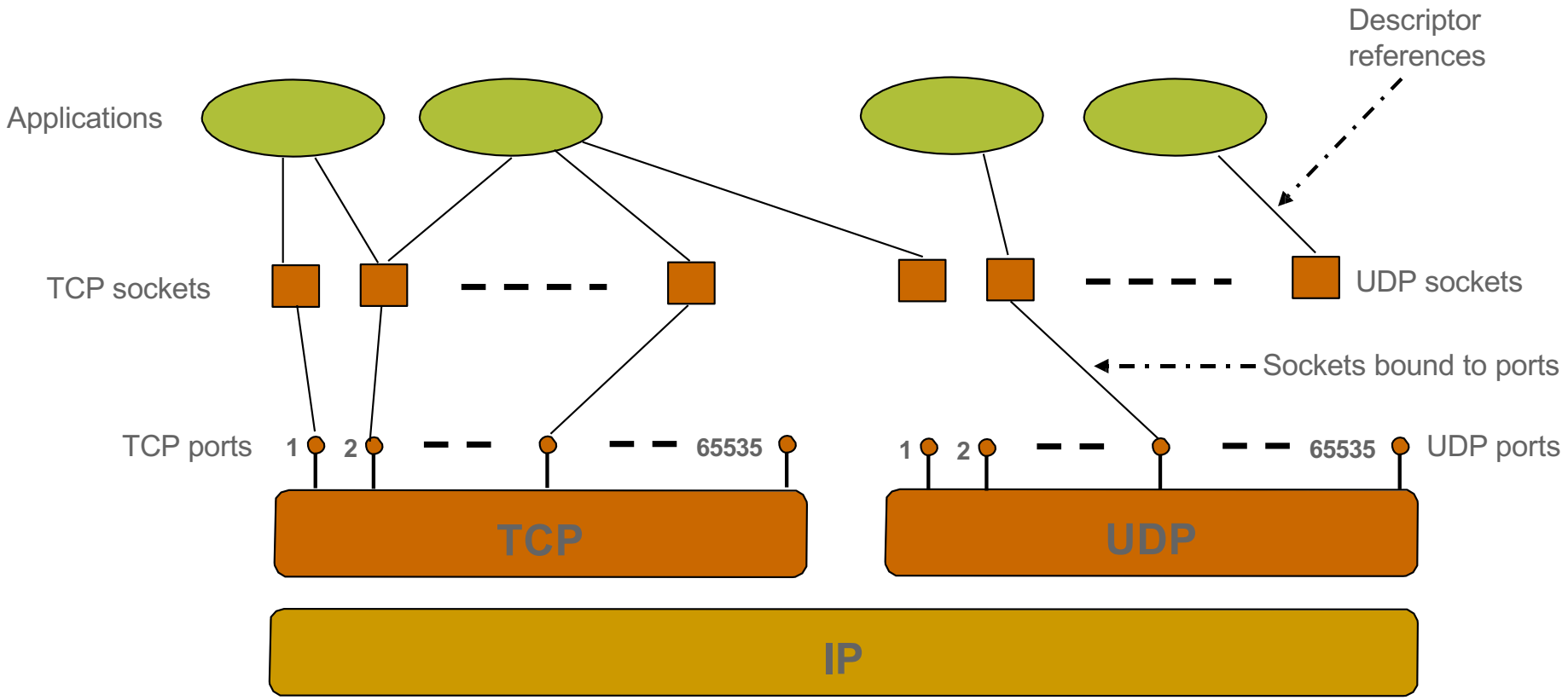


Sockets

- Uniquely identified by
  - an internet address
  - an end-to-end protocol (e.g. TCP or UDP)
  - a port number
- Two types of (TCP/IP) sockets
  - Stream sockets (e.g. uses TCP)
    - provide reliable byte-stream service
  - Datagram sockets (e.g. uses UDP)
    - provide best-effort datagram service
    - messages up to 65.500 bytes
- Socket extend the convectional UNIX I/O facilities
  - file descriptors for network communication
  - extended the read and write system calls



Sockets


Descriptor
references
Applications
TCP sockets
UDP sockets
Sockets bound to ports
TCP ports
1
2
65535
1
2
65535
UDP ports
TCP
UDP
IP

Socket programming

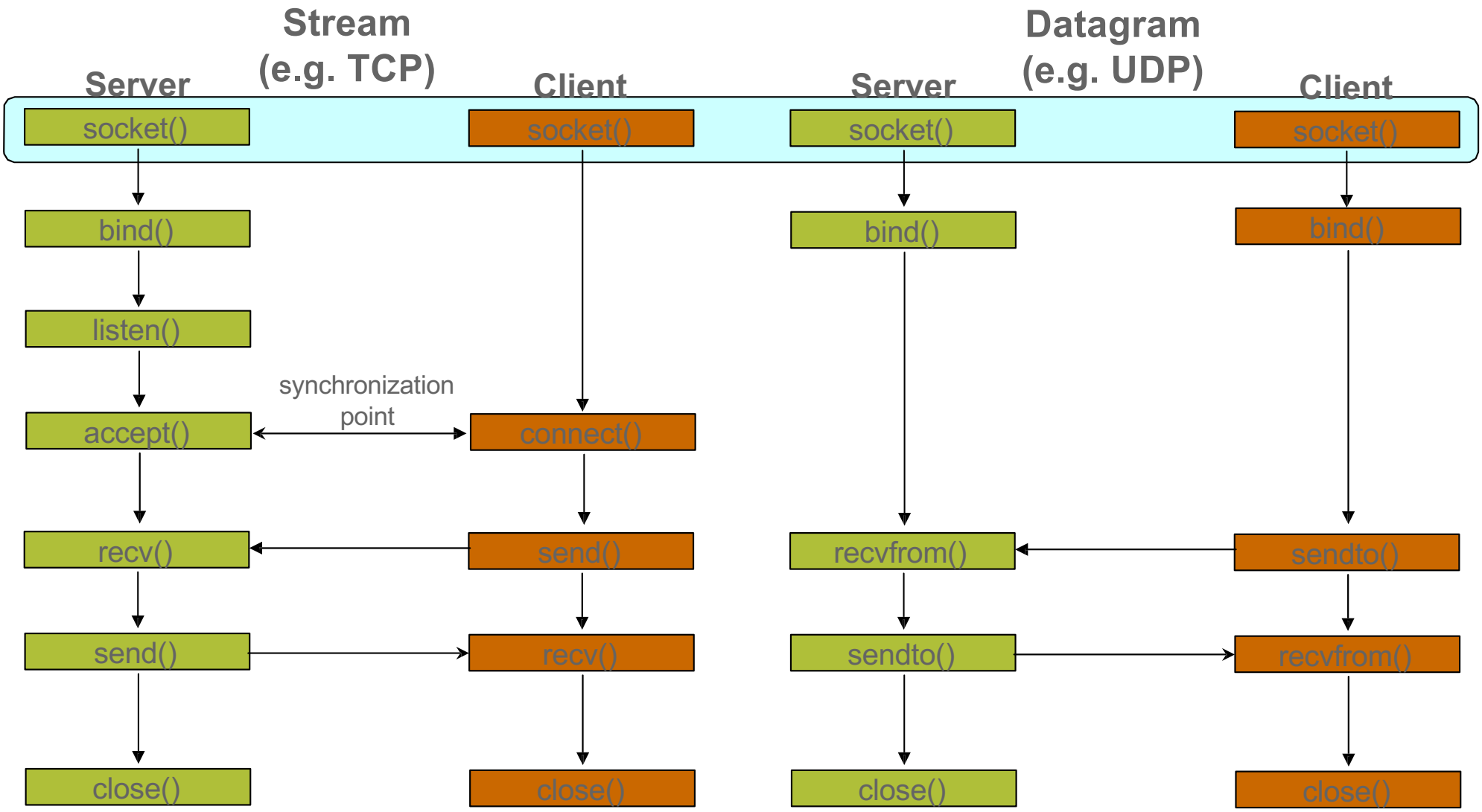
Client-server communication

- Server
  - passively waits for and responds to clients
  - passive socket
- Client
  - initiates the communication
  - must know the address and the port of the server
  - active socket

Sockets - Procedures

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Client - Server Communication



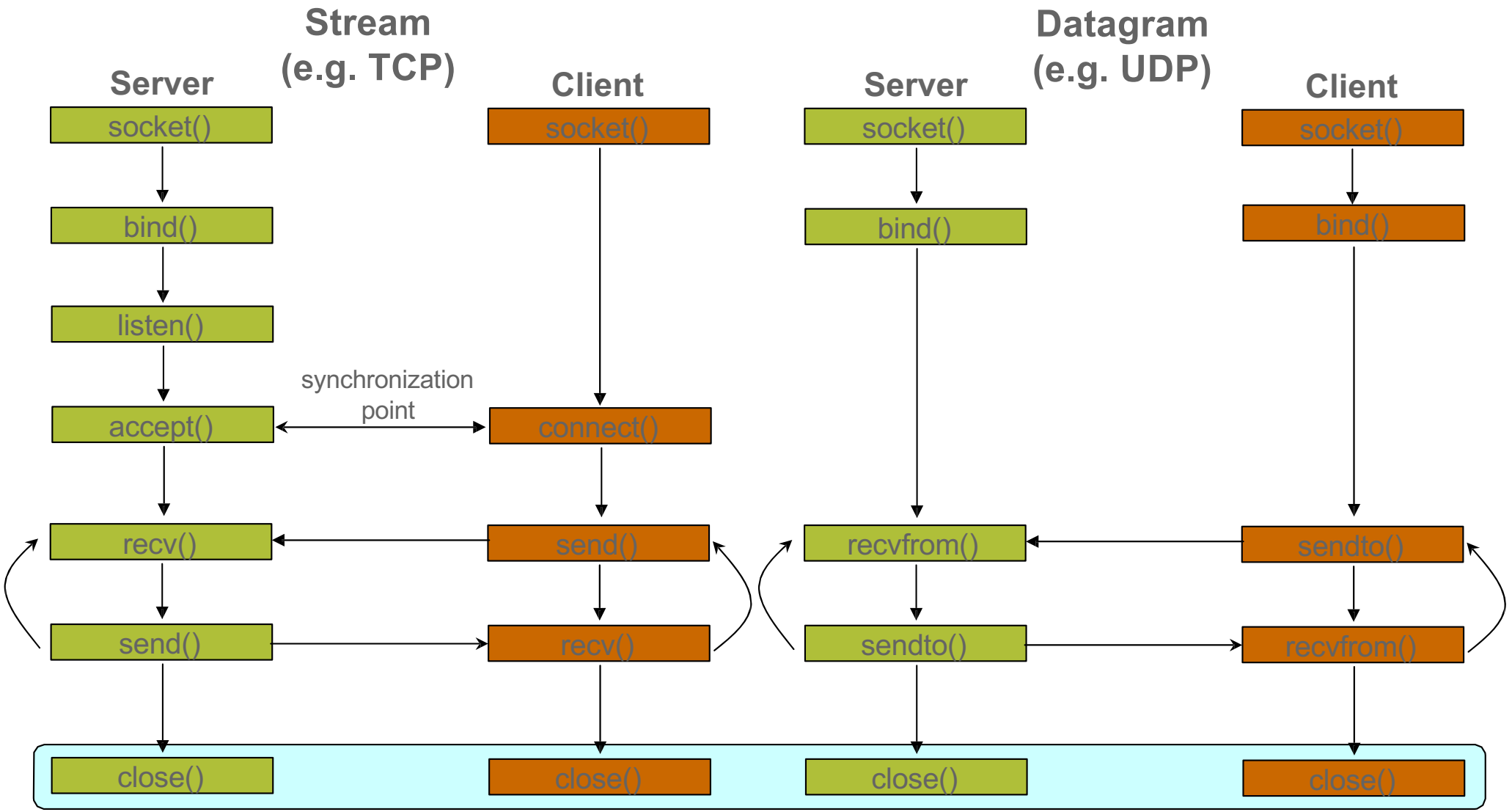
Socket creation in C: socket()

```
int sockid = socket(family, type, protocol);
```

- **sockid**: socket descriptor, an integer (like a file-handle)
- **family**: integer, communication domain, e.g.,
  - PF_INET, IPv4 protocols, Internet addresses (typically used)
  - PF_UNIX, Local communication, File addresses
- **type**: communication type
  - SOCK_STREAM - reliable, 2-way, connection-based service
  - SOCK_DGRAM - unreliable, connectionless, messages of maximum length
- **protocol**: specifies protocol
  - IPPROTO_TCP IPPROTO_UDP
  - usually set to 0 (i.e., use default protocol)
- upon failure returns -1

NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

Client - Server Communication



Socket close in C

- When finished using a socket, the socket should be closed

```
status = close(sockid);
```

- **sockid**: the file descriptor (socket being closed)
- **status**: 0 if successful, -1 if error

- Closing a socket
  - closes a connection (for stream socket)
  - frees up the port used by the socket

Specifying Addresses

Socket API defines a **generic** data type for addresses:

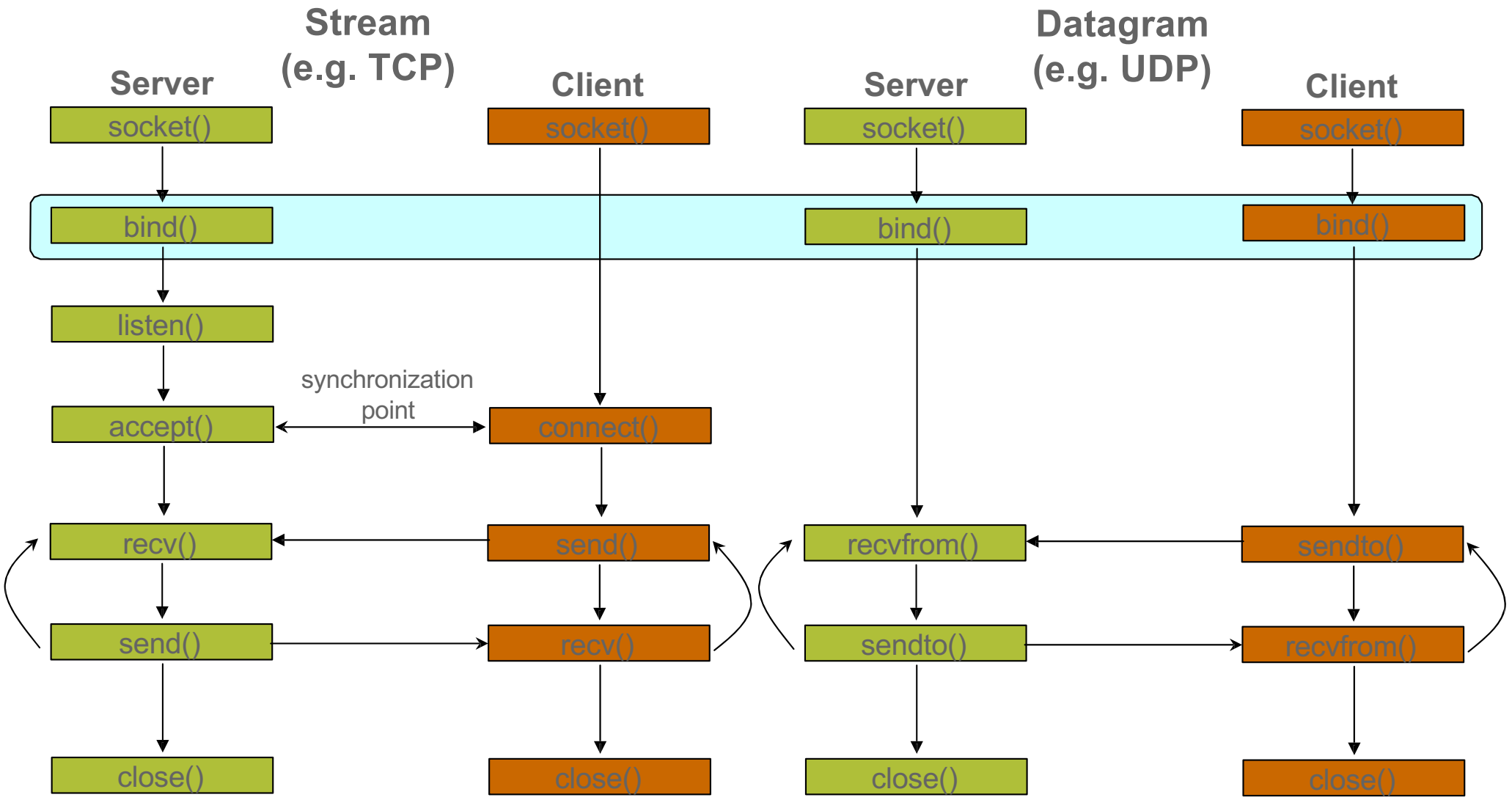
```
struct sockaddr {
    unsigned short sa_family;  /* Address family (e.g. AF_INET) */
    char sa_data[14];          /* Family-specific address information */
}
```

Particular form of the sockaddr used for **TCP/IP** addresses:

```
struct in_addr {
    unsigned long s_addr;          /* Internet address (32 bits) */
}

struct sockaddr_in {
    unsigned short sin_family;     /* Internet protocol (AF_INET) */
    unsigned short sin_port;       /* Address port (16 bits) */
    struct in_addr sin_addr;       /* Internet address (32 bits) */
    char sin_zero[8];              /* Not used */
}
```

Important: sockaddr_in can be casted to a sockaddr

Client - Server Communication



Assign address to socket

- associates and reserves a port for use by the socket

```
int status = bind(sockid, &addrport, size);
```

- **sockid**: integer, socket descriptor
- **addrport**: struct sockaddr, the (IP) address and port of the machine
  - for TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming interface
- **size**: the size (in bytes) of the addrport structure
- **status**: upon failure -1 is returned

bind()- Example with TCP

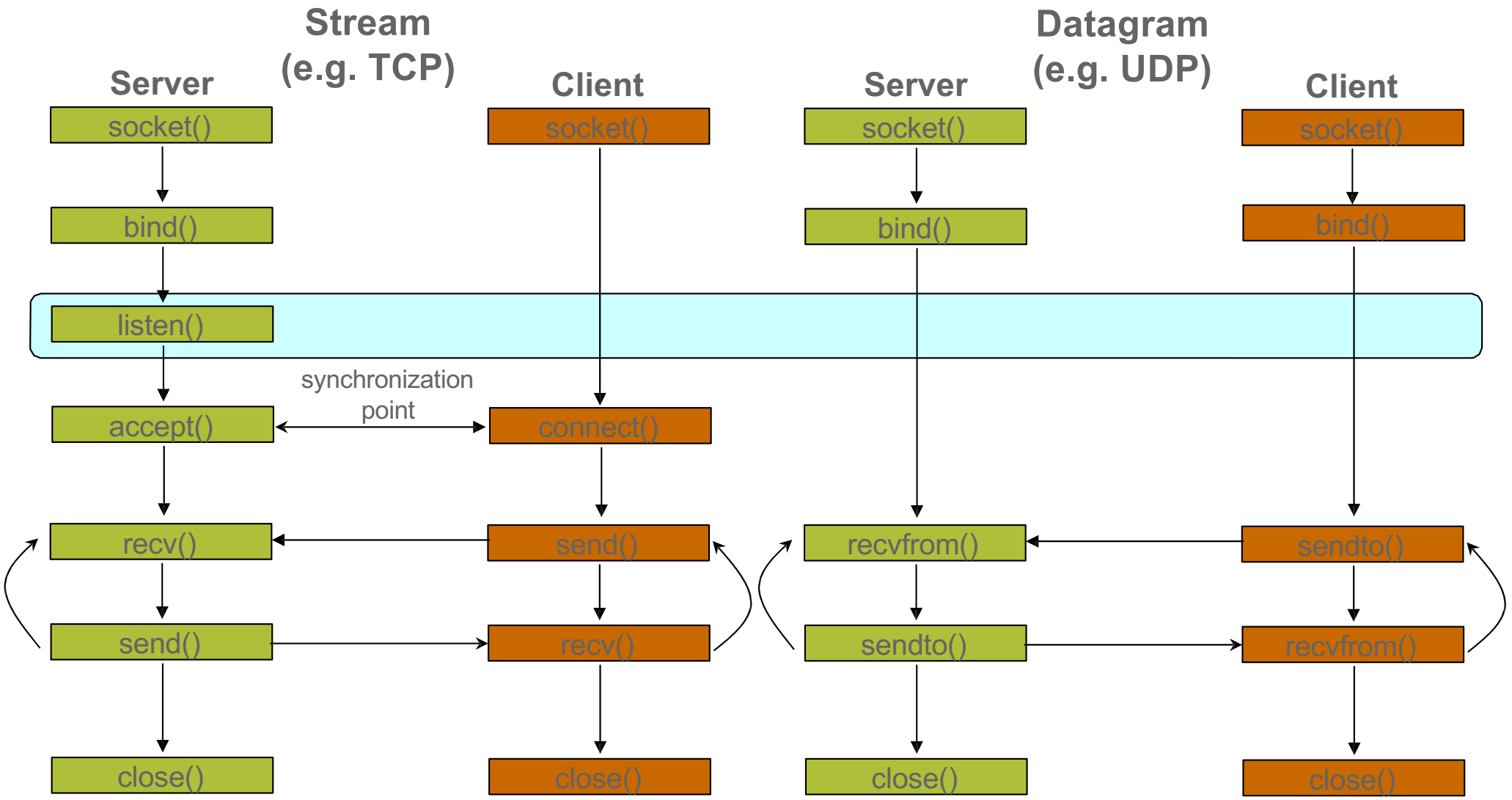
```
int sockid;
struct sockaddr_in addrport;
sockid = socket(PF_INET, SOCK_STREAM, 0);

addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sockid, (struct sockaddr *) &addrport, sizeof(addrport))!= -1) {
   …}
```

Skipping the bind()

- bind can be skipped for both types of sockets

- Datagram socket:
  - if only sending, no need to bind.
    - The OS finds a port each time the socket sends a packet
  - if receiving, need to bind
- Stream socket:
  - destination determined during connection setup
  - don’t need to know port sending from (during connection setup, receiving end is informed of port)

Client - Server Communication



listen

Instructs TCP protocol implementation to listen for connections

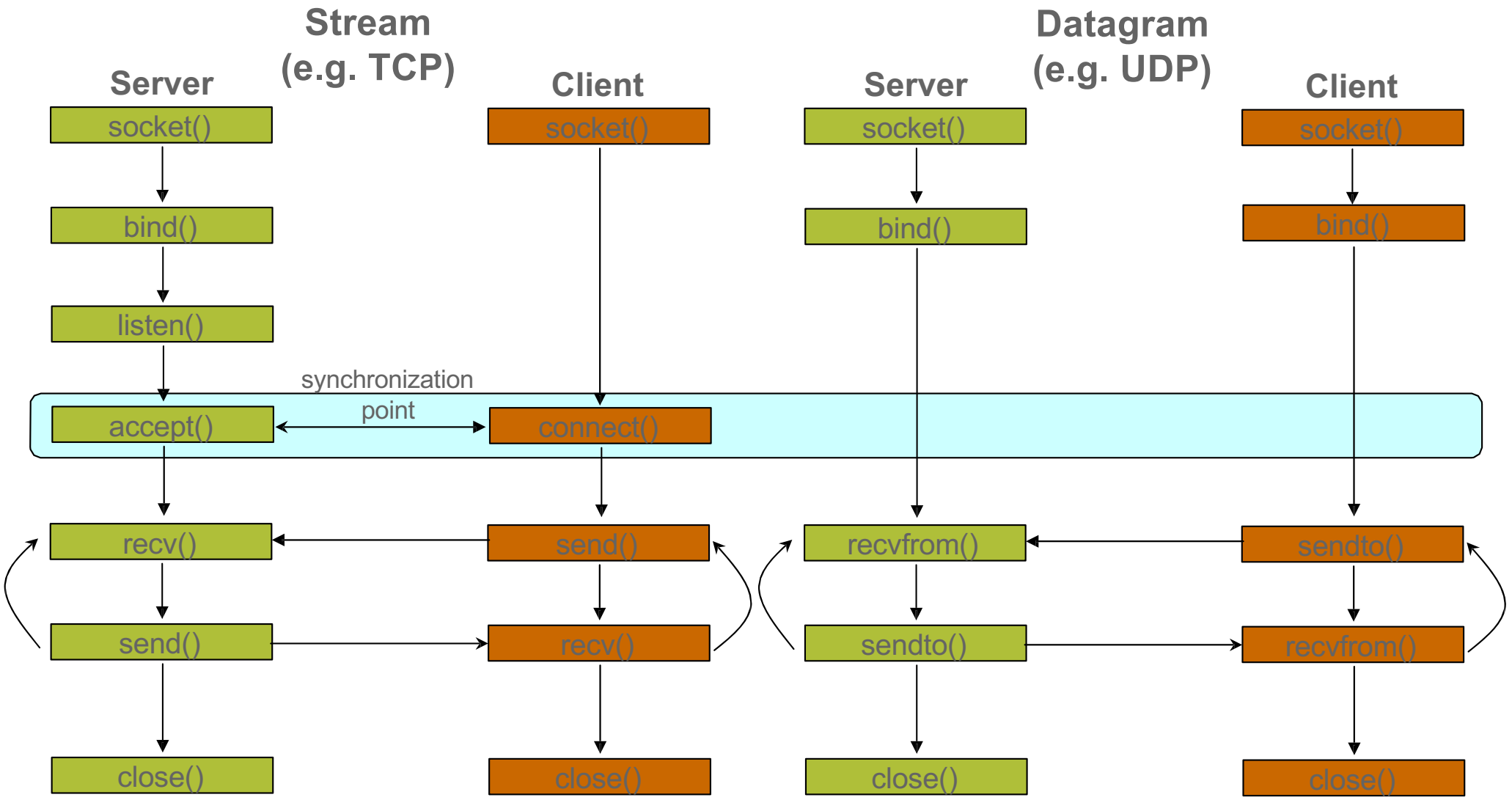
```
int status = listen(sockid, queueLimit);
```

- **sockid**: integer, socket descriptor
- **queuelen**: integer, # of active participants that can "wait" for a connection
- **status**: 0 if listening, -1 if error

- listen()is **non-blocking**: returns immediately

- The listening socket (sockid)
  - is never used for sending and receiving
  - is used by the server only as a way to get new sockets

Client - Server Communication



Establish Connection: connect()

- The client establishes a connection with the server by calling connect()

```
int status = connect(sockid, &foreignAddr, addrlen);
```

- **sockid**: integer, socket to be used in connection
- **foreignAddr**: struct sockaddr: address of the passive participant
- **addrlen**: integer, sizeof(name)
- **status**: 0 if successful connect, -1 otherwise

- Important: connect() is blocking

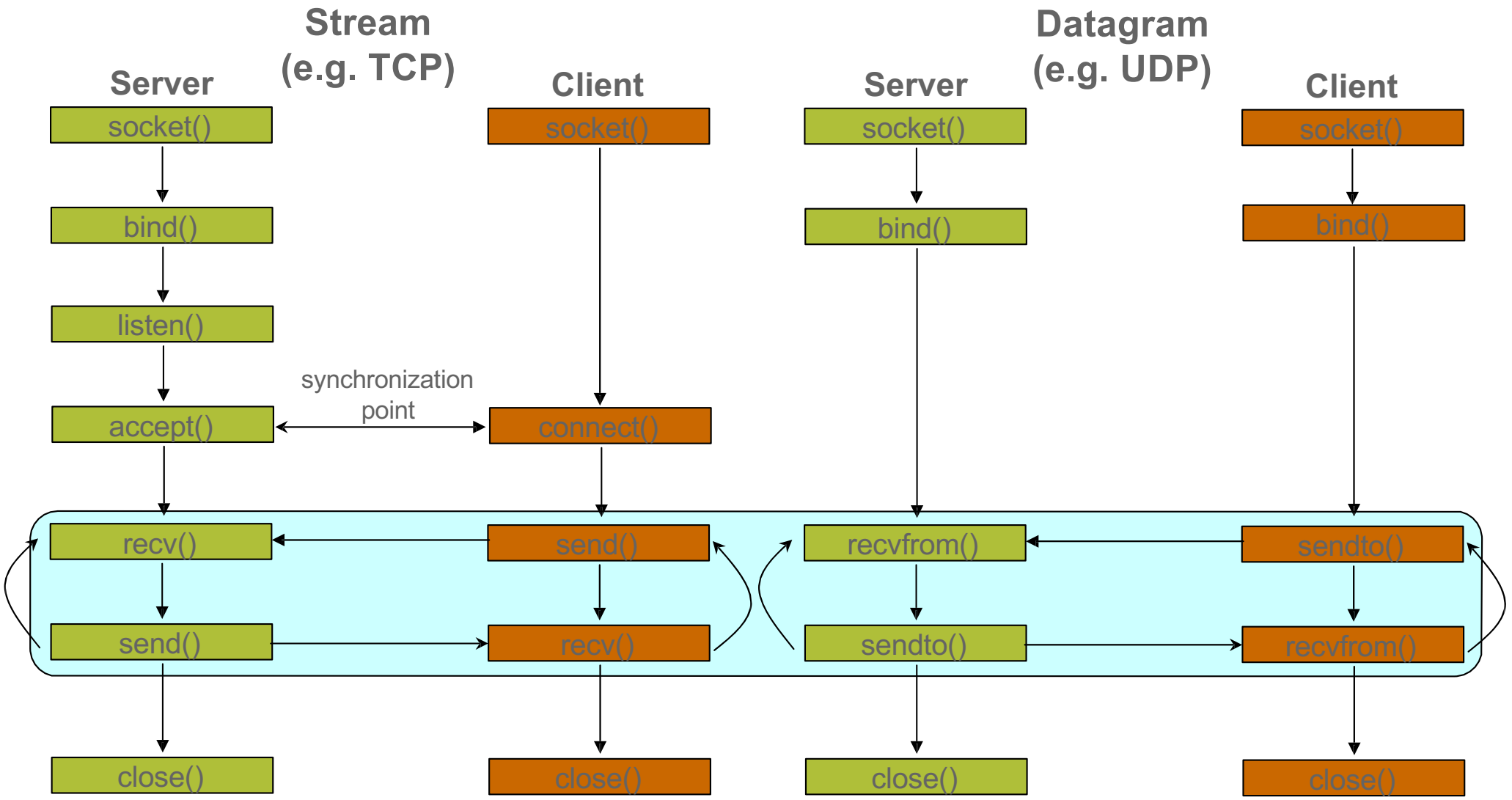
Establish Connection: accept()

- The server gets a socket for an incoming client connection by calling accept()

```
int s= accept(sockid, &clientAddr, &addrLen);
```

- **s**: integer, the new socket (used for data-transfer)
- **sockid**: integer, the orig. socket (being listened on)
- **clientAddr**: struct sockaddr, address of the active participant
  - filled in upon return
- **addrLen**: sizeof(clientAddr): value/result parameter
  - must be set appropriately before call
  - adjusted upon return
- accept()
  - is **blocking**: waits for connection before returning
  - dequeues the next connection on the queue for socket (sockid)

Client - Server Communication



Exchanging data with stream socket

int count = send(sockid, msg, msgLen, flags);

- msg: const void[], message to be transmitted
- msgLen: integer, length of message (in bytes) to transmit
- flags: integer, special options, usually just 0
- count: # bytes transmitted (-1 if error)

int count = recv(sockid, recvBuf, bufLen, flags);

- recvBuf: void[], stores received bytes
- bufLen: # bytes received
- flags: integer, special options, usually just 0
- count: # bytes received (-1 if error)
- Calls are blocking
  - returns only after

Exchanging data with datagram socket

```
int count = sendto(sockid, msg, msgLen, flags, &foreignAddr, addrlen);
```

- msg, msgLen, flags, count: same with send()
- foreignAddr: struct sockaddr, address of the destination
- addrLen: sizeof(foreignAddr)

```
int count = recvfrom(sockid, recvBuf, bufLen, flags, &clientAddr, addrlen);
```

- recvBuf, bufLen, flags, count: same with recv()
- clientAddr: struct sockaddr, address of the client
- addrLen: sizeof(clientAddr)
- Calls are blocking
  - returns only after data is sent / received

Exercise

- Implement the garden exercise in a networked based way by using sockets
  - E.g. the counter is implemented as part of server and the turnstile is implemented as clients.

Overview

I. Introduction - parallel architectures
II. Distributed Architectures
III. **GPU architecture and programming**

Architecture

CPU = Central Processing Units

- single or miltiple processing units (cores)
- standalone or integrated into clusters
- designed to run processes, supports threads

GPU = Graphical Processing Units

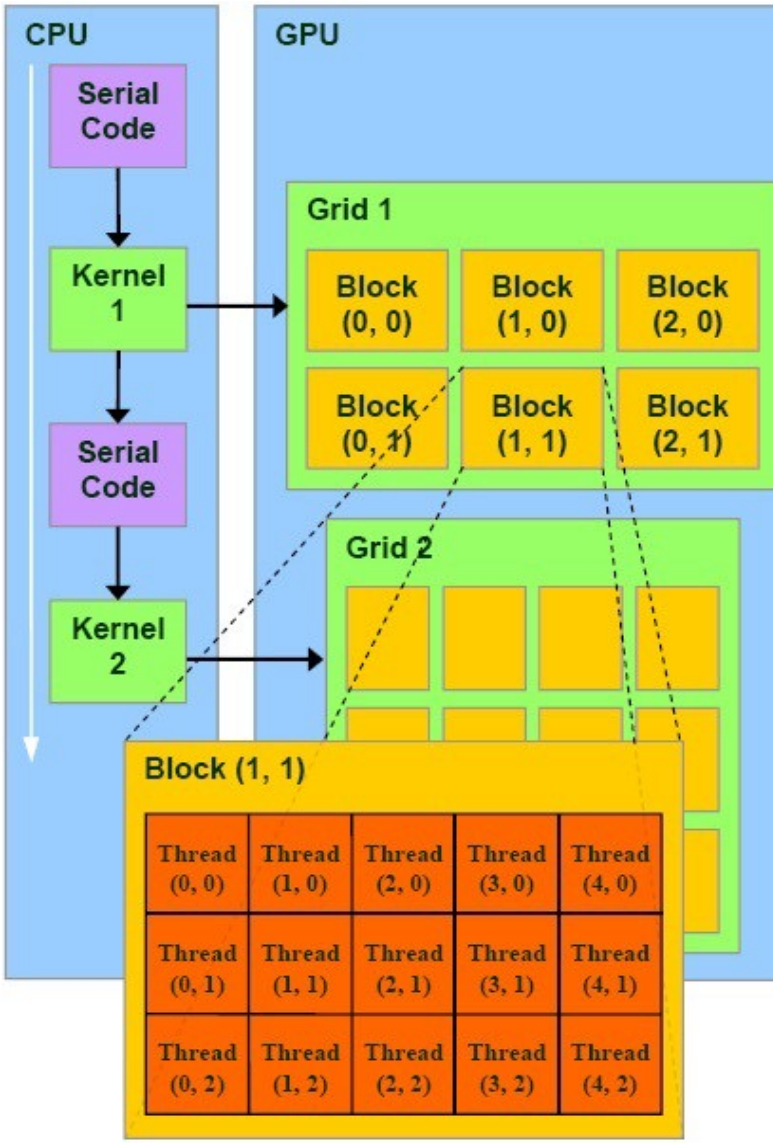
- Usually attached to a host CPU
- Developed for games, and visualization (OpenGl, think Pixar)
- Designed to run lightweight threads
- Accessible via specialized libraries, compiler directives (OpenACC), and extensions to languages (C, C++ and Fortran).

CUDA (Compute Unified Device Architecture)

- a parallel computing platform and programming model created by NVIDIA.
- extension of C programming language

G□□ A□□□□□□□□□□□□□

- GPU is a highly threaded coprocessor to the host CPU and associated memory
- Kernels are sections of the application that are run on the GPU by a thread.
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
  - Each block is organized as 3D array of threads: (*blockDim.x*, *blockDim.y*, and *blockDim.z*)
- Threads within a thread block must:
  - execute the same kernel
  - share data, so they must be issued to the same processor
- A grid is a collection of blocks:
  - A Grid is organized as a 2D array of blocks: (*gridDim.x* and *gridDim.y*)



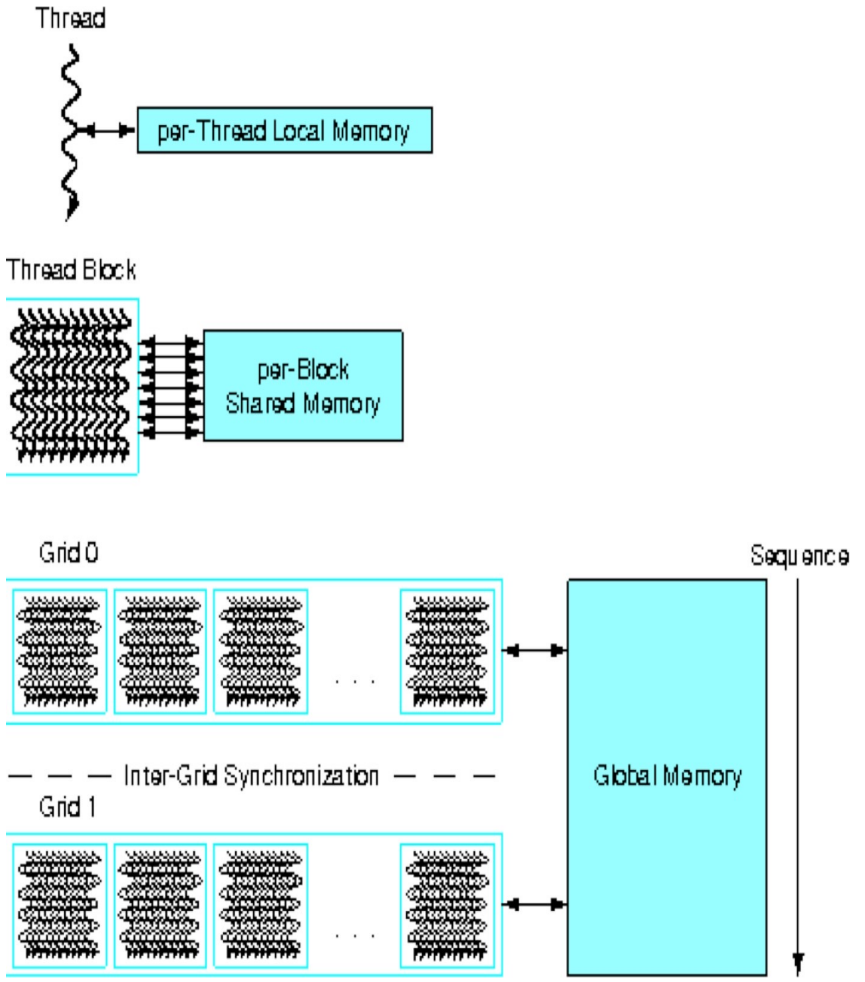
Source: http://hothardware.com/Articles/NVIDIA-GF100-Architecture-and-Feature-Preview

GPU Kernel

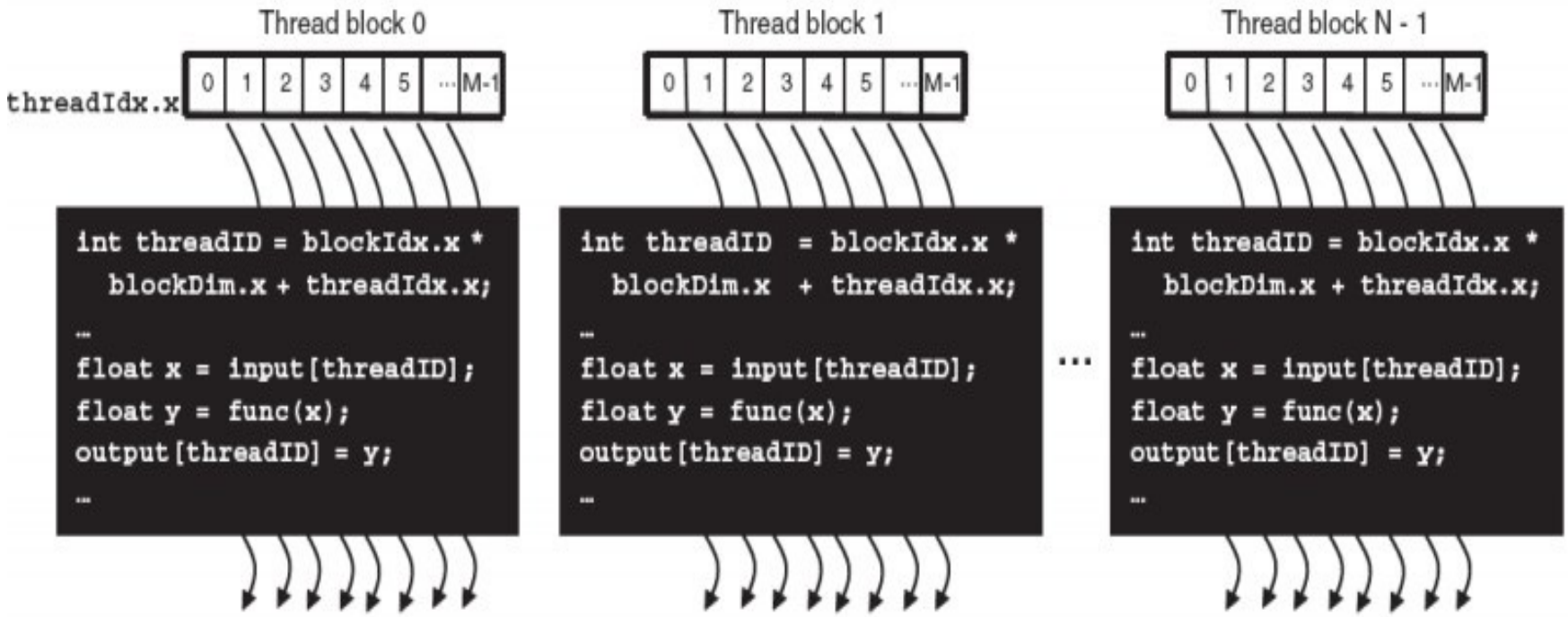
- Kernels are not full applications:
  - they are the parallel sections or critcal blocks
- They are executed by a grid of unordered thread blocks:
  - Till 2010 max 512 threads per block -> now >512 * x
  - Thread blocks start at the same instruction address, execute in parallel
  - Blocks can have different endpoints (divergence) but these are limited
  - Communicate through shared memory and synchronization barriers
  - Must be assigned to the same (stream) processor

G□□: □□□□□□□□

- A thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler.
- A thread is a light-weight process.
- In most cases, a thread is contained inside a process.
- Multithreading generally occurs by time-division multiplexing (as in multitasking)
- Multiprocessor (including multi-core system): threads or tasks run at the same time - each processor or core runs a particular thread or task.



NVIDIA Thread Calculations



- Thread ID is unique within a block
- Block ID is unique
- Can make unique ID for each thread per kernel using Thread and Block IDs.

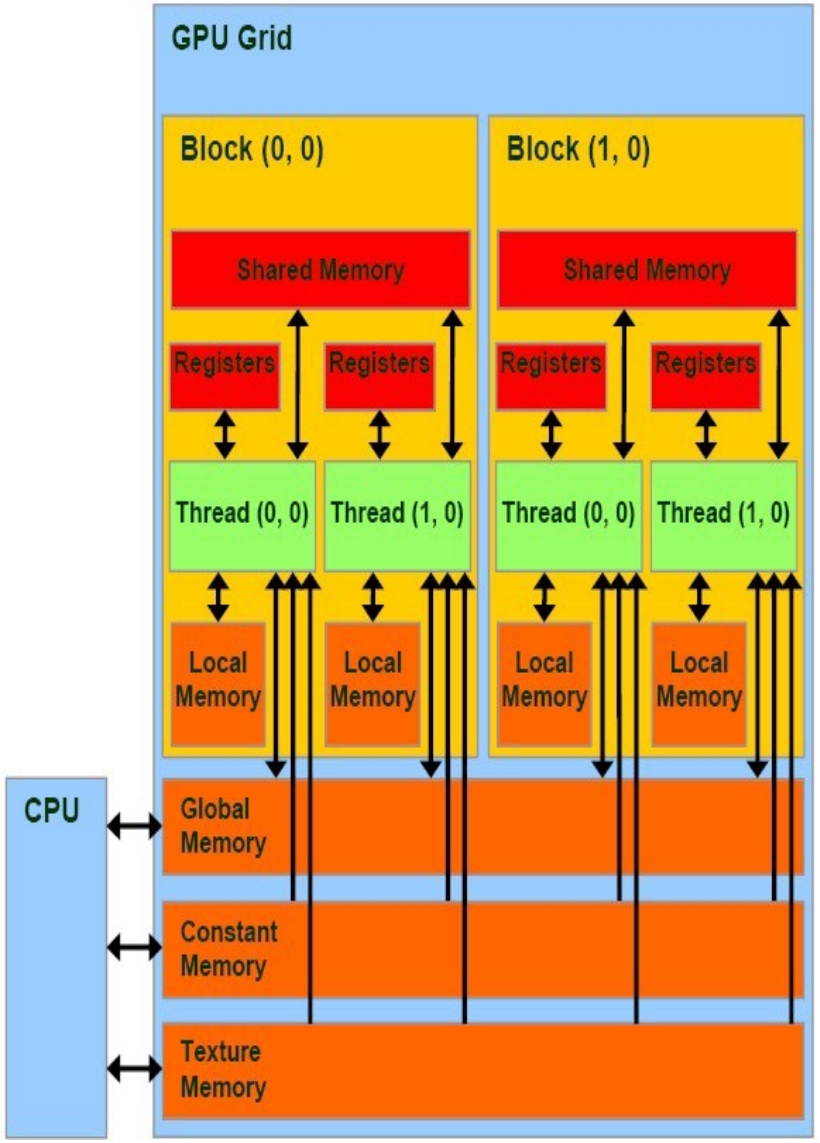
G□□: G□□□

- A collection of blocks that can (but are not required to) execute in parallel.
- Theres no synchronization at all between the blocks.
- Number of [concurrent] grids on a GPU: … eg. nvidia

  1 for GPU Cores < 2.0
  16 for 2.0 <= CC <= 3.0
  32 for CC = 3.5
  . . .
  Need to use right API in order to avoid serialization.
- Use API for querying the GPU system.

GPU Memory

- Red is fast on-chip, orange is DRAM
- Register file & local memory are private for each thread
- Shared memory is used for communication between threads (appx same latency as regs)
- DRAM:
  - Constant memory used for random accesses (such as instructions)
  - Texture memory (large) and has two dimensional locality
- Global Memory: visible to an entire grid, can be arbitrarily written to and read from by the GPU or the CPU.

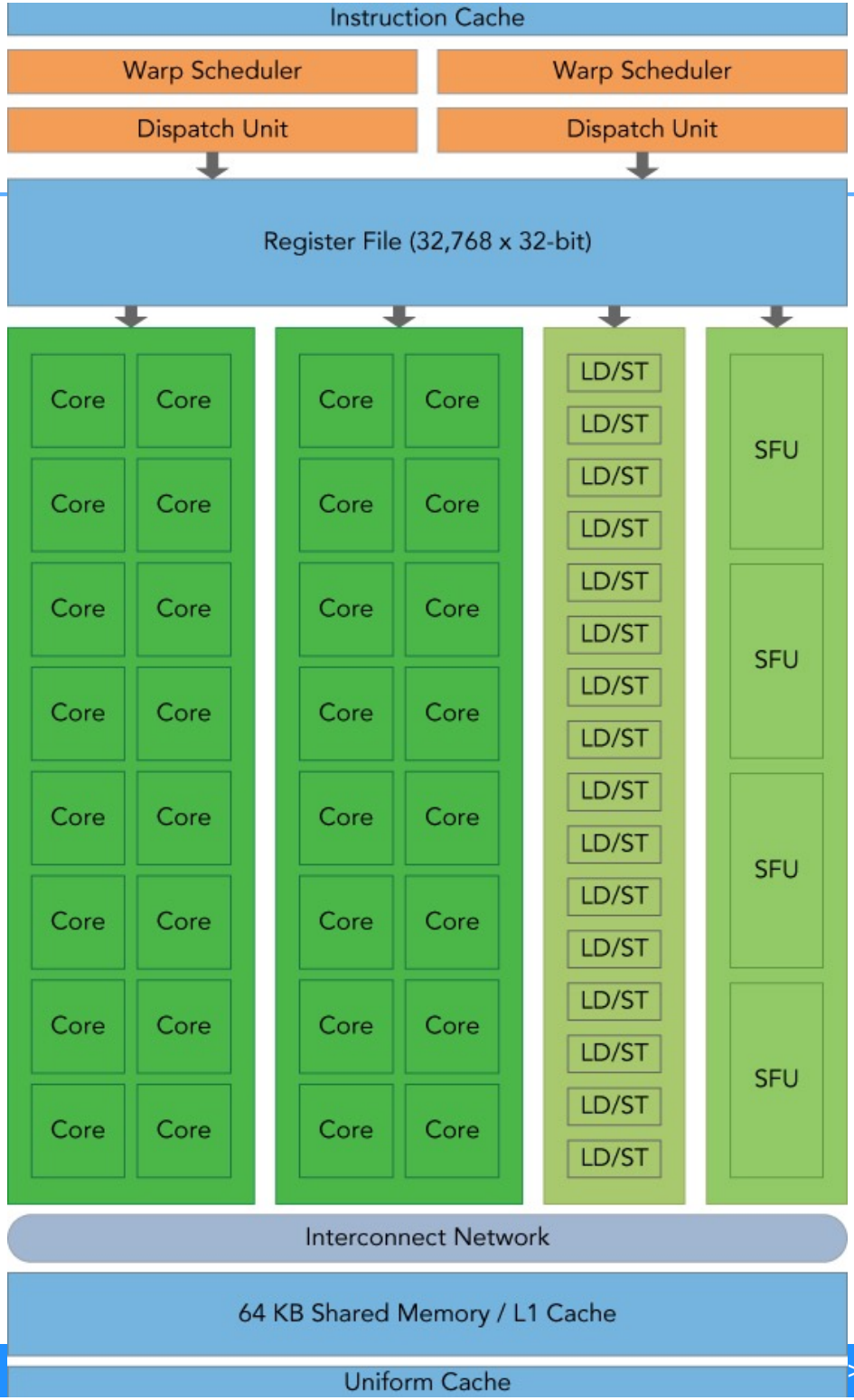


Source: NVIDIA

Key components

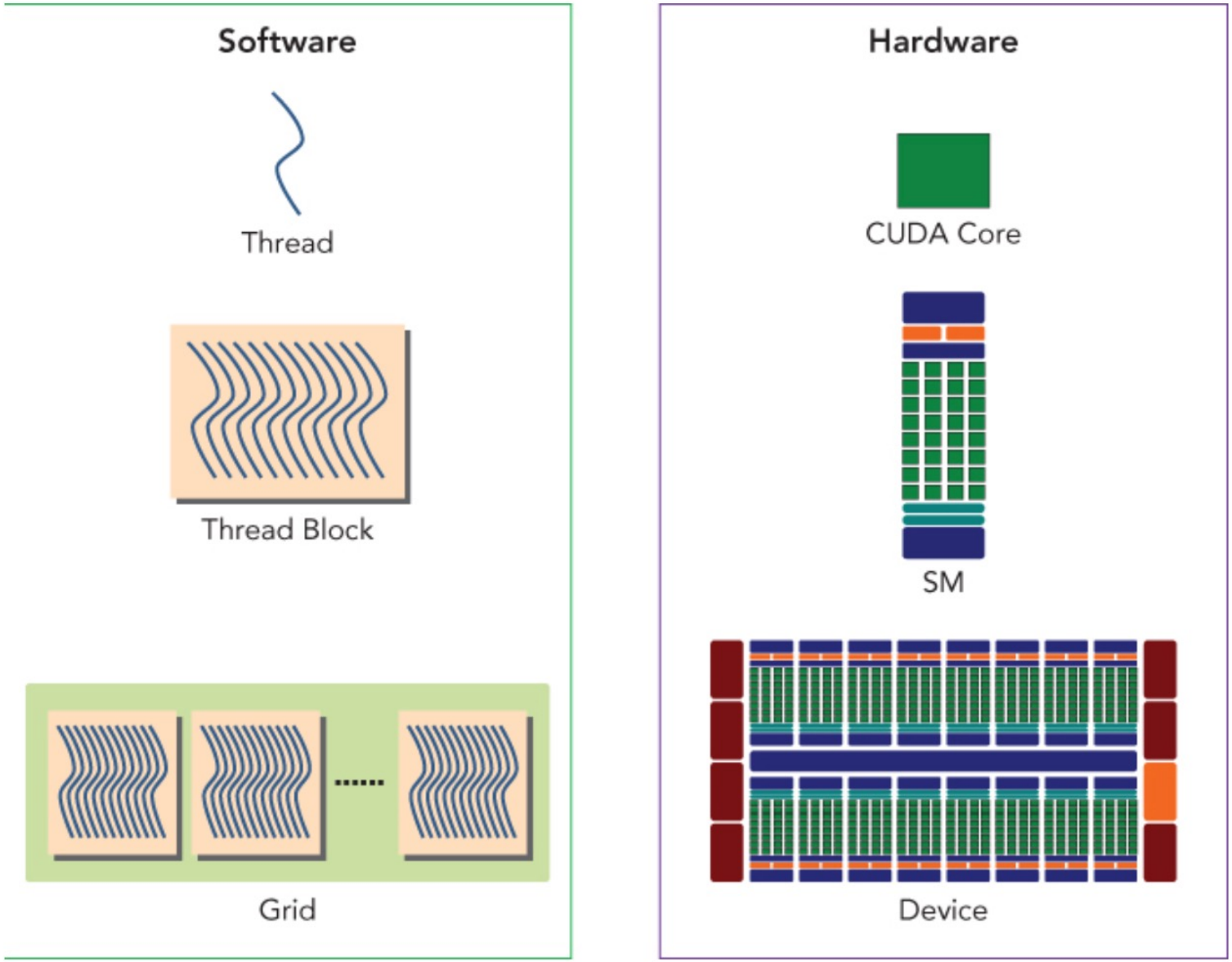
Subheadline

- SFU
  - Special function unit
  - Like square root, interpolation
- LD/ST
  - Load store unit

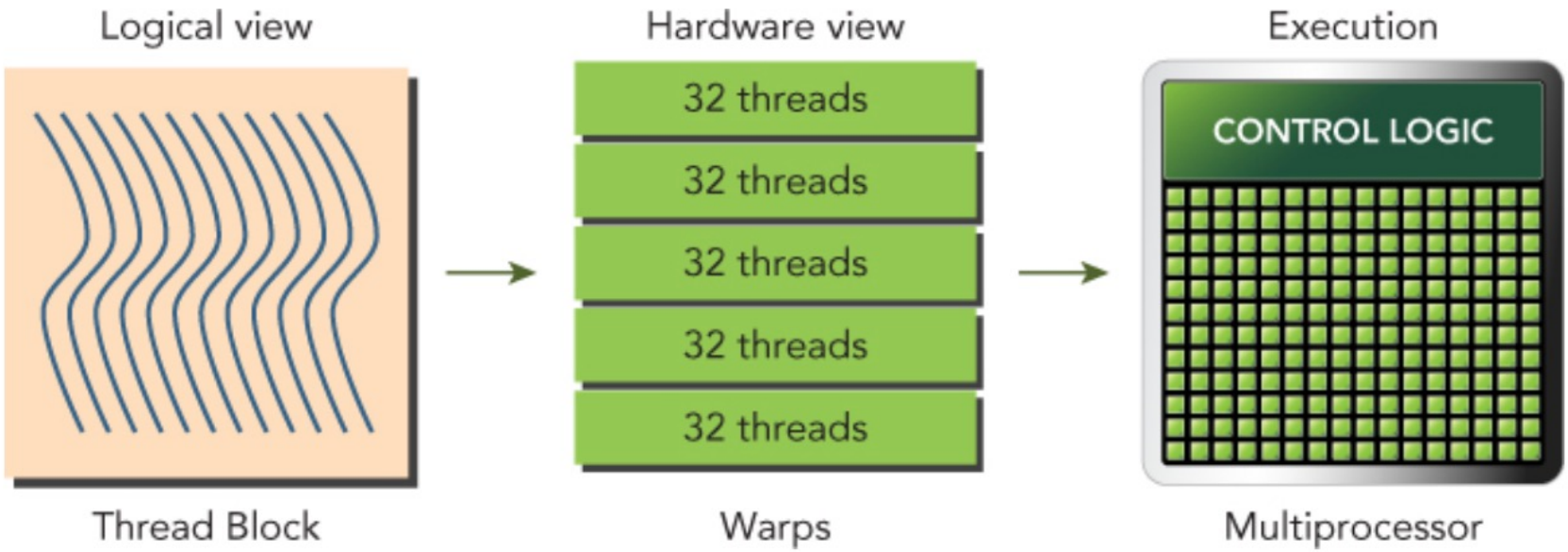


SW and HW View

Subheadline

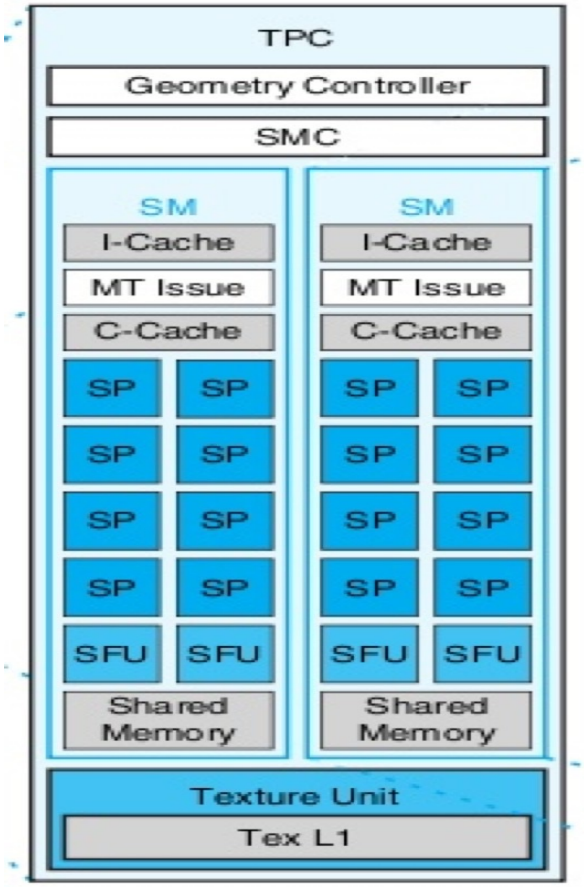


Relationship between logical view and hardware view of a thread block



□□D A H□□□□I□□□:GEF□□□□8800

- Streaming Multiprocessors (SMs, also called nodes)
- 8 Stream Processors (SPs) (or cores): primary thread processor
- has 1000's of registers that can be partitioned among threads of execution
- Multiple caches:
  - shared memory for fast data interchange between threads,
  - constant cache for fast broadcast of reads from constant memory,
  - texture cache to aggregate bandwidth from texture memory,
  - L1 cache: reduce latency to memory
- warp schedulers: switch contexts between threads and instructions to warps;
- Execution cores:
  - Integer and floating point ops
  - Special Function Units (SFUs)

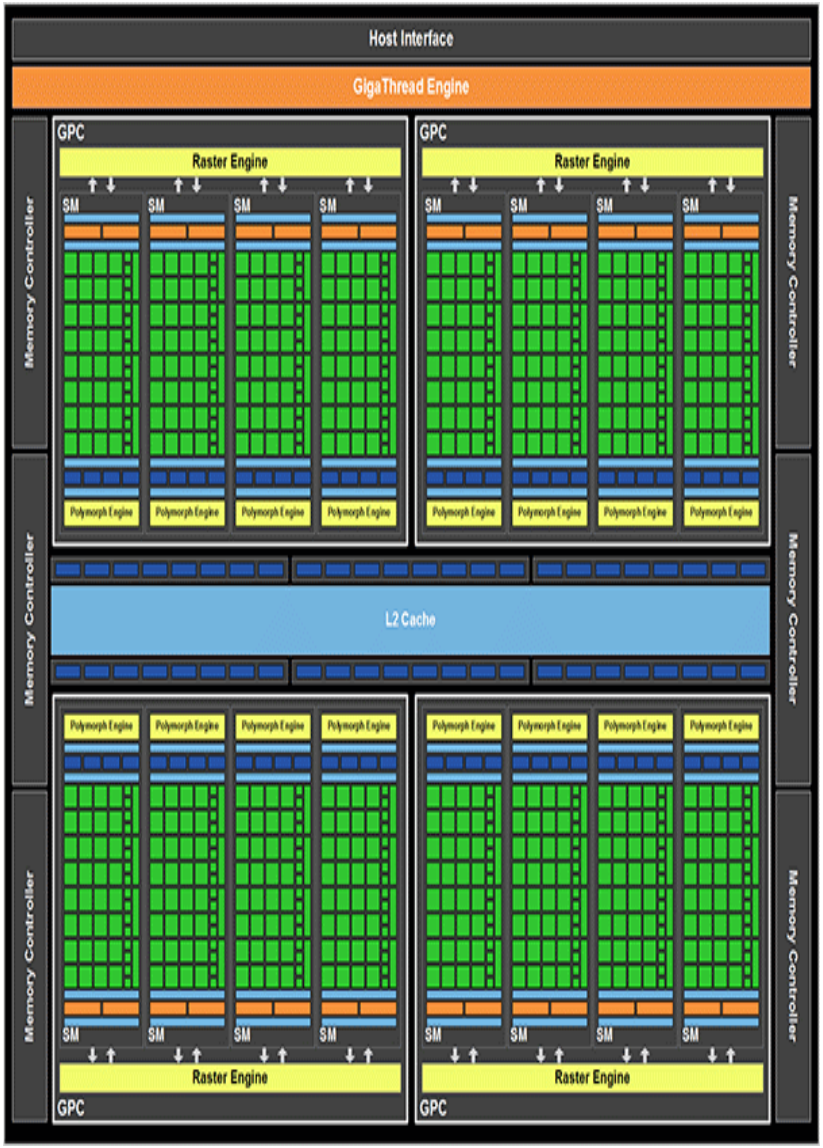


□□IDIA G□□ GF100 H□□-□□□□□B□□□D□□□□□□ (2010)

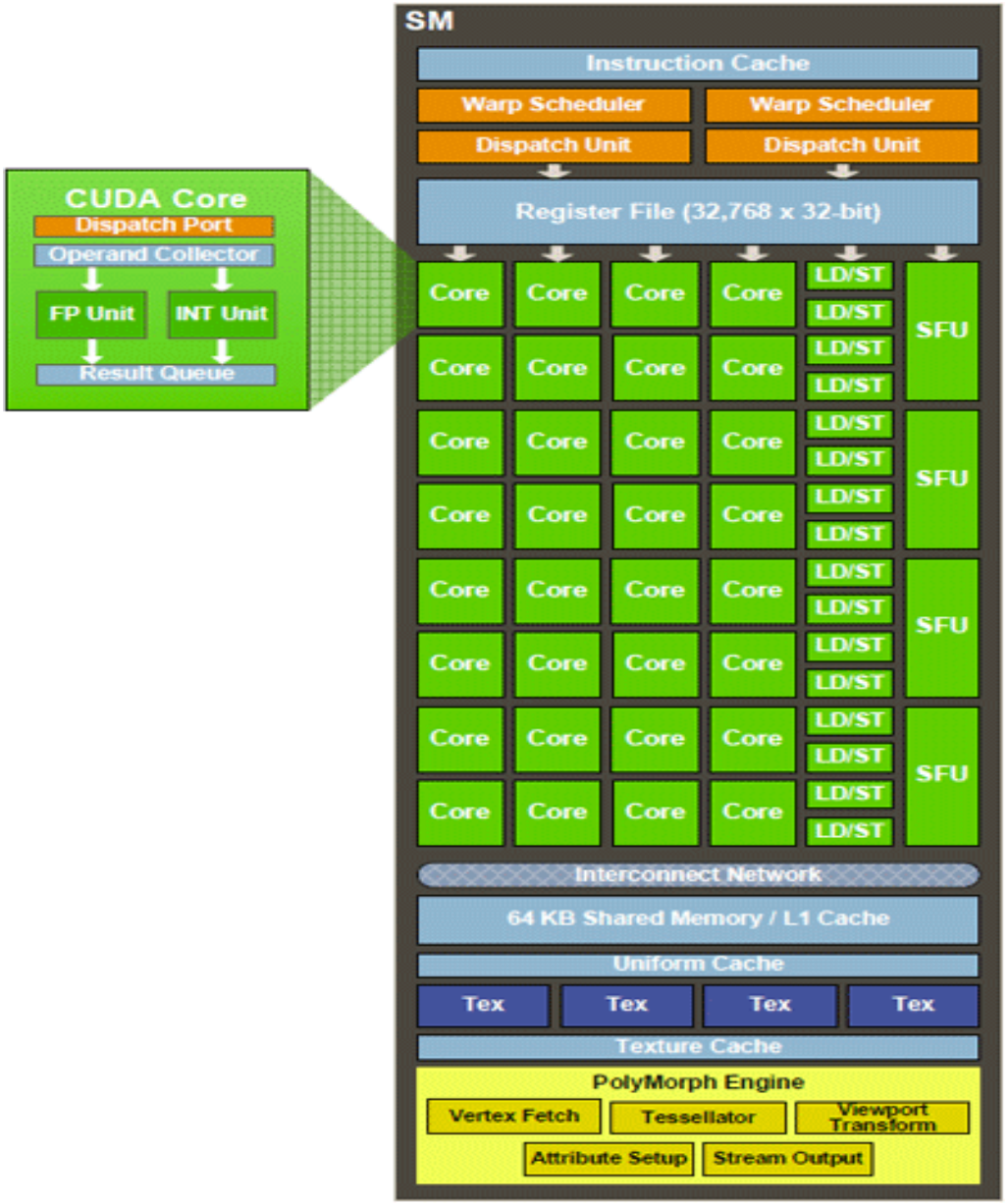
- CPU is the host
- GPU is the device
- The GF100 has 4 Graphics Processing Clusters (GPCs): laid out in (2x2) arrangement (also called ”Raster Engine”).
- Each GPC has 4 Streaming Multiprocessors, (SMs): NVIDIAs’ term for multiprocessor (also called ”Polymorph Engines”).
  - Arranged in 1x4 layout.
  - Total number of SMs = 4 ∗4 = 16
- Each SM has a block of Stream Processors (SPs) or Cores– also called execution units.
  - Arranged in 8x4 layout.
  - Total number of SPs on each SM = 8 ∗4 = 32

Total number of cores on the GPU

$$\#Cores = \#SPs/SM \times \#SMs/GPC \times \#GPCs$$
$$= 32 \times 4 \times 4 = 512$$

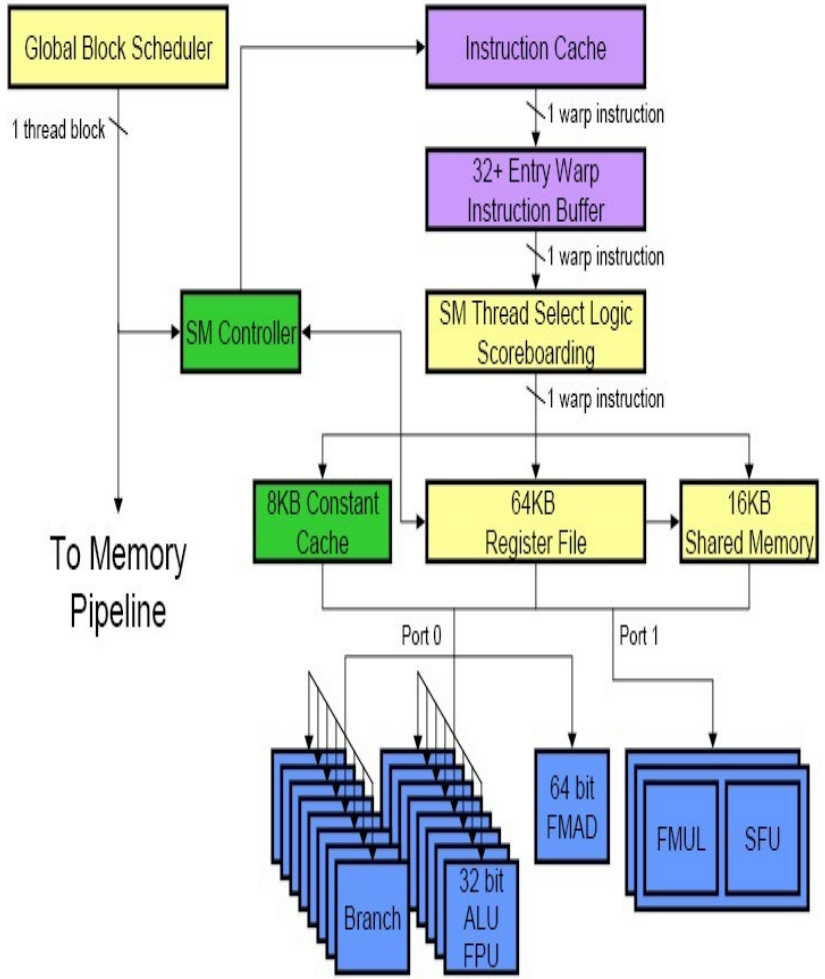


- each SM block in each GPC is comprised of 32 cores
- 48/16KB of shared memory (3 x that of GT200),
- 16/48KB of L1 (there is no L1 cache on GT200),



□□D□A G□200 □□ A□□□(2008)

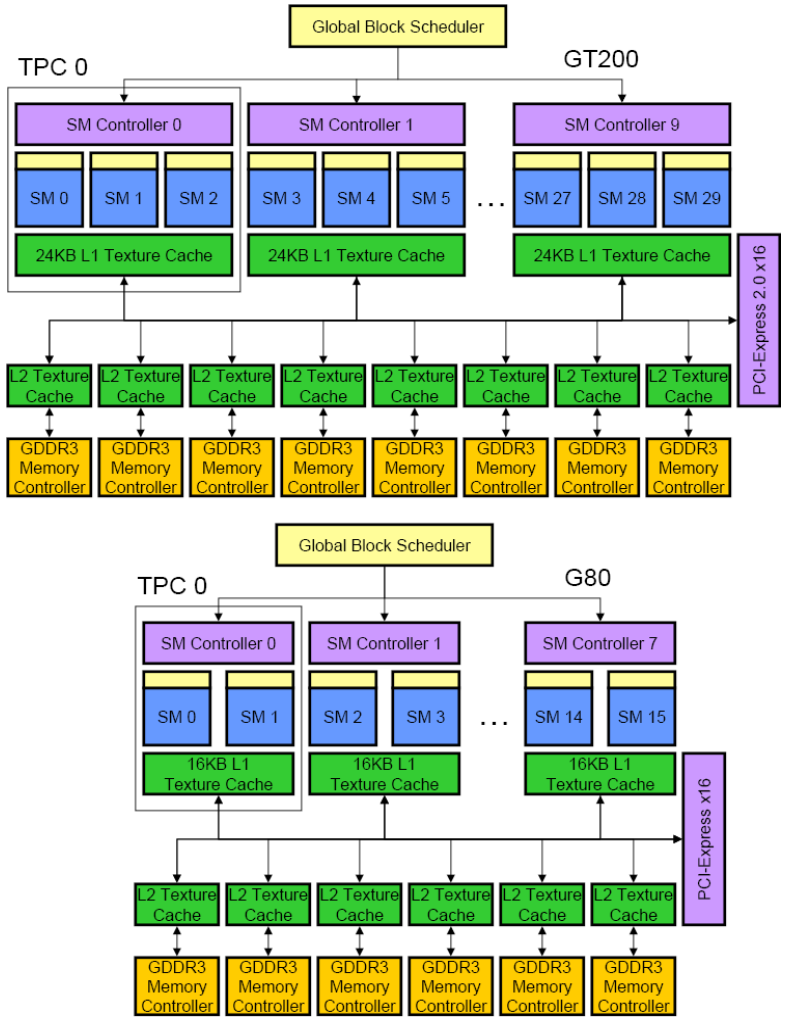
- highly threaded single-issue processor with SIMD/SIMT (single instruction multiple thread)
- 8 functional units
- Each SM can execute up to 8 thread blocks concurrently and a total of 1024 threads concurrently
- warp: a group of threads managed by SM thread scheduler
- Single Instruction, Multiple Thread (SIMT) programming model



Source: http://www.realworldtech.com/gt200

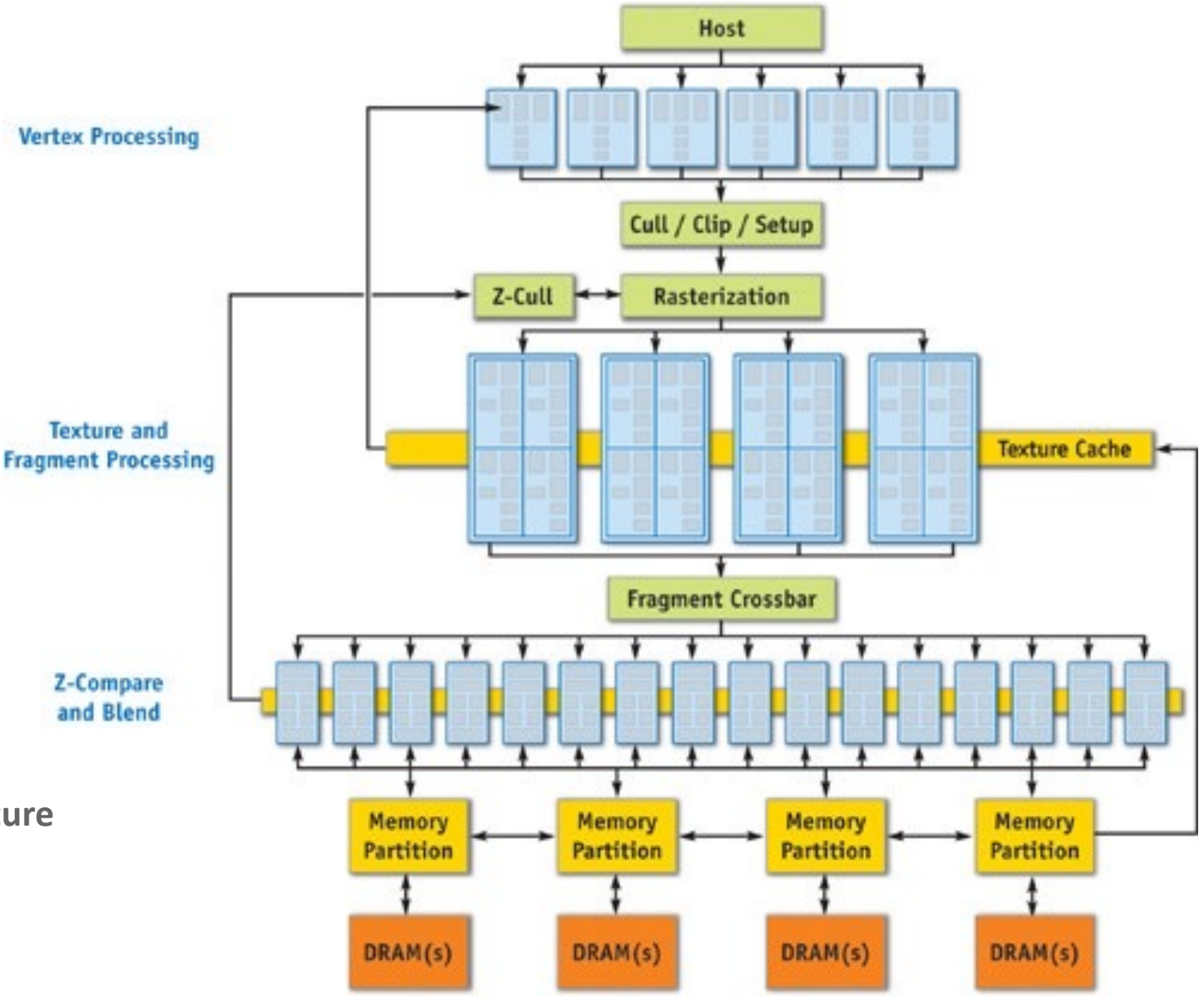
GPU Global Scheduler (work distribution unit)

- Manages coarse grained parallelism at thread block level
- At kernel startup, information for grid sent from CPU (host) to GPU (device)
- Scheduler reads information and issues thread blocks to streaming multiprocessors (SM)
- Issues thread blocks in a round-robin fashion to SMs
- Uniformly distribute threads to SMs
- Key distribution factors:
  - kernel demand for threads per block
  - shared memory per block
  - registers per thread
  - thread and block state requirements
  - current availability resources in SM



http://www.realworldtech.com/gt200/6/

Specialized Pipeline Architecture



GeForce 6 Series Architecture

(2004-5)

From GPU Gems 2

A100

- https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

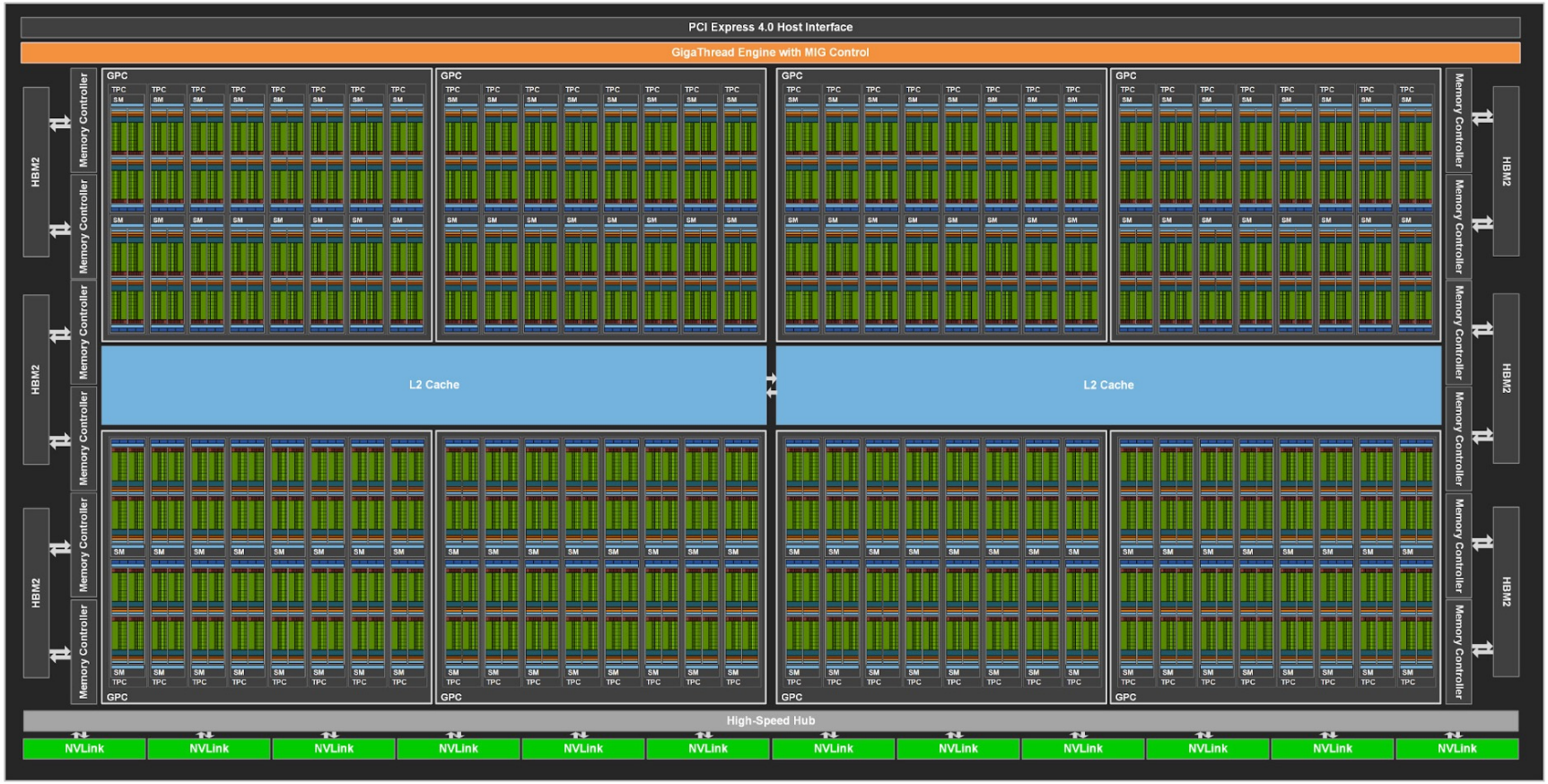


Figure 6. GA100 Full GPU with 128 SMs (A100 Tensor Core GPU has 108 SMs)

Figure 7. GA100 Streaming Multiprocessor (SM)

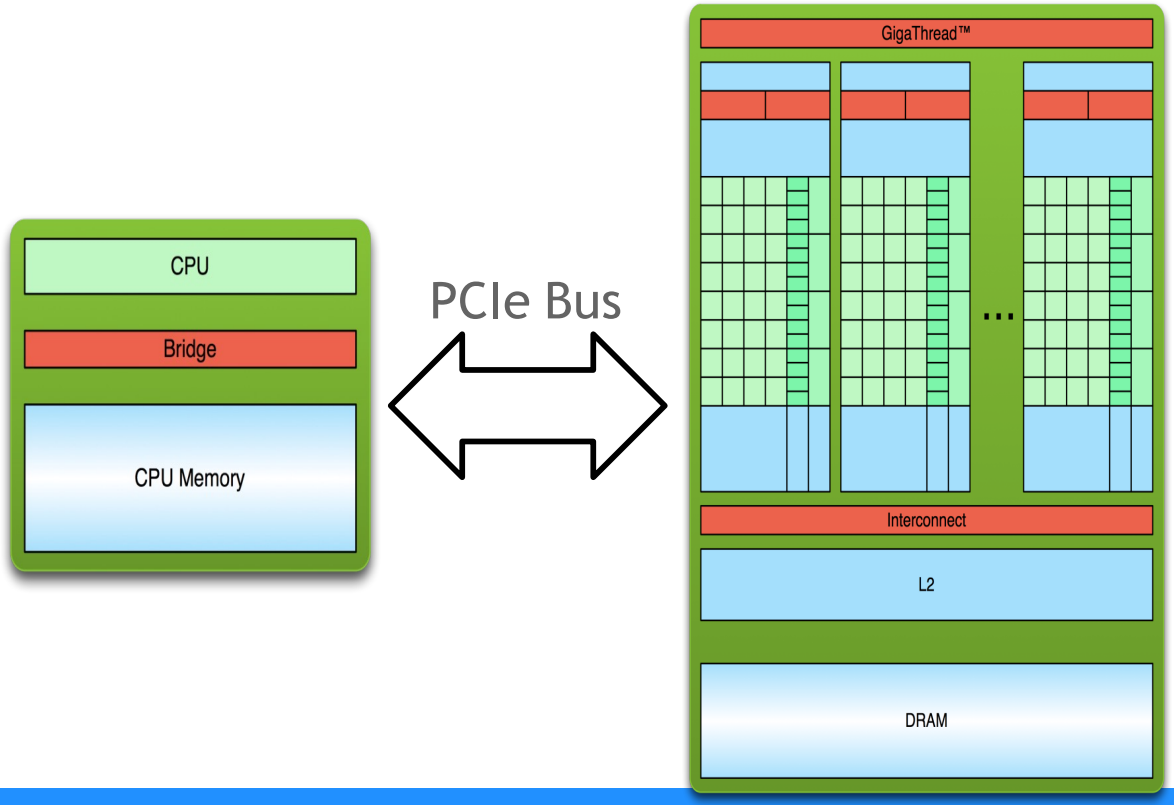
Parallelism in CPUs v. GPUs

- CPUs use ***task parallelism***
  - Multiple tasks map to multiple threads
  - Tasks run different instructions
  - Less number of relatively heavyweight threads run less number of cores
  - Each thread managed and scheduled explicitly
  - Each thread has to be individually programmed
- GPUs use ***data parallelism***
  - SIMD model (Single Instruction Multiple Data)
  - Same instruction on different data
  - 10,000s/* of lightweight threads on 100s/* of cores
  - Threads are managed and scheduled by hardware
  - Programming done for batches of threads (e.g. one pixel shader per group of pixels, or draw call)

The Host and the GPU

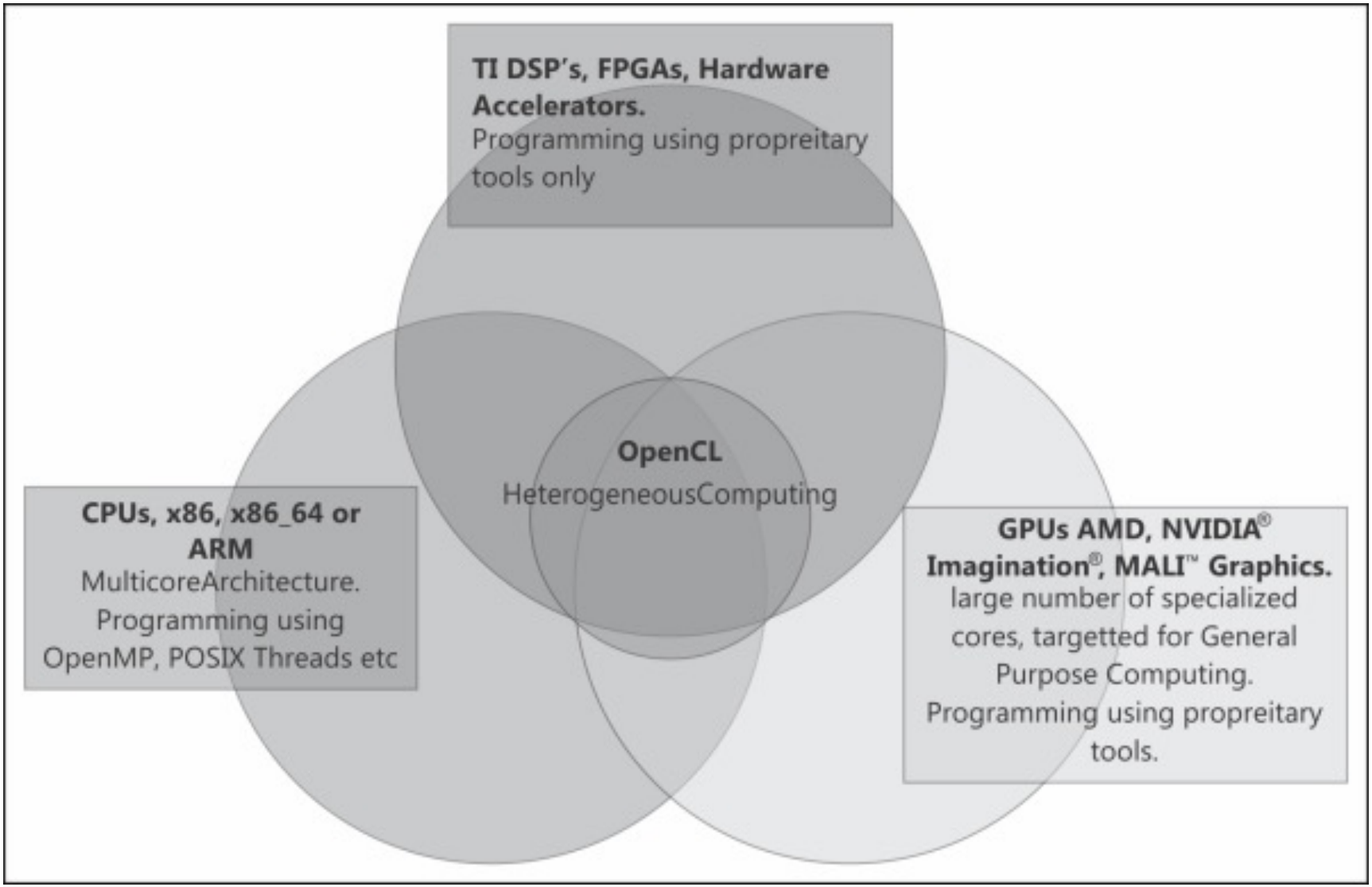
- Today, the GPU is not a standalone processor

- GPUs are always tied to a management processing unit, commonly referred to as the host, and always a CPU

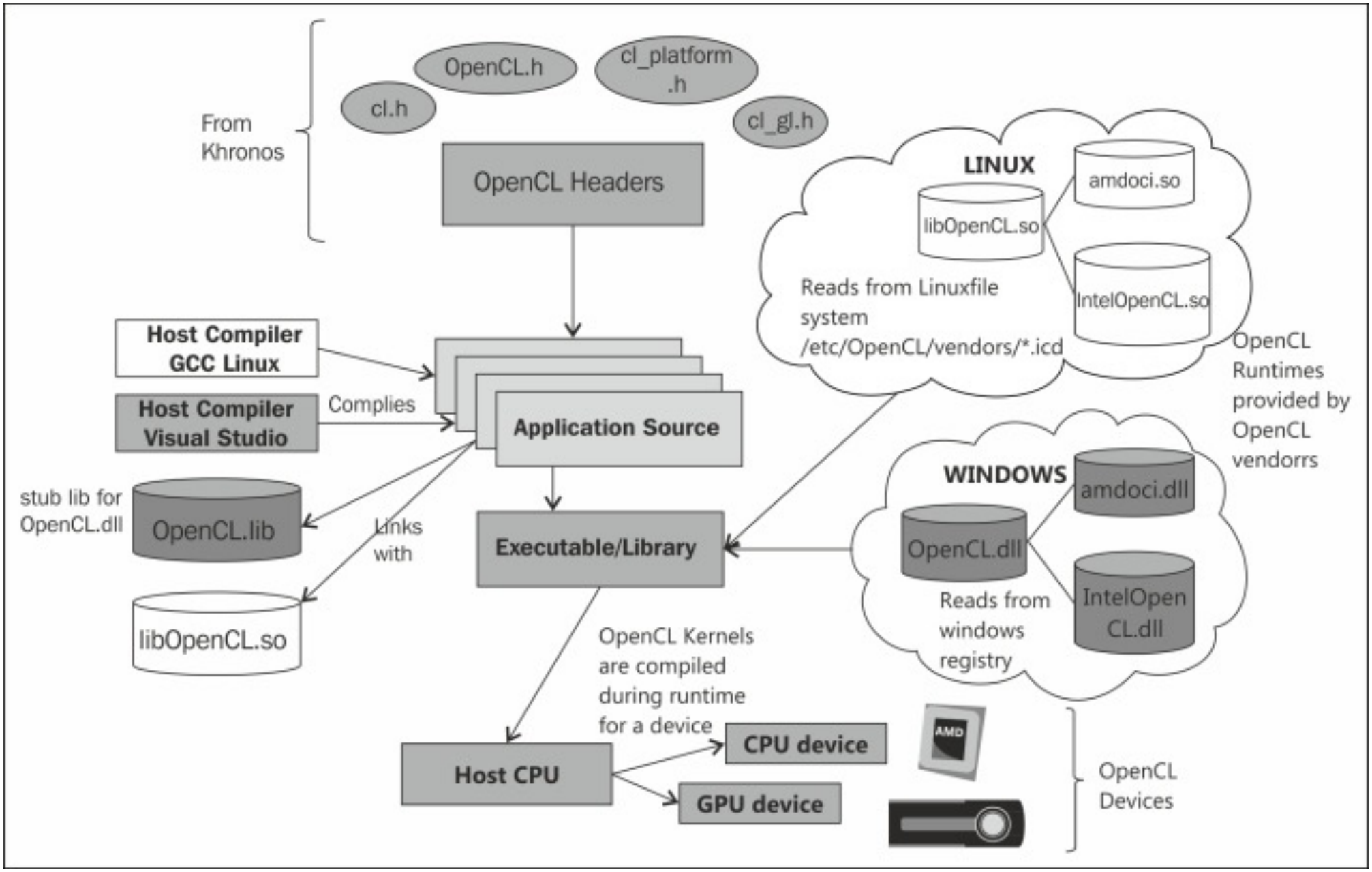


OpenCL

Heteregenous systems



OpenCL components



Setting up Opencl platforms

Running OSX?

- OpenCL works out of the box!
- Just compile your programs with

```
-framework OpenCL -DAPPLE
```

Setting up with AMD GPU

- Install some required packages:
  - `sudo apt-get install build-essential linux-headers-generic debhelper dh-modaliases execstack dkms lib32gcc1 libc6-i386 opencl-headers`
- Download the driver from amd.com/drivers
  - Select your GPU, OS, etc.
  - Download the .zip
  - Unpack this with unzip filename.zip
- Create the installer
  - `sudo sh fglrx*.run --buildpkg Ubuntu/precise`
- Install the drivers
  - `sudo dpkg -i fglrx*.deb`
- Update your Xorg.conf file
  - `sudo amdconfig --initial --adapter=all`
- Reboot!
  - Check all is working by running `fglrxinfo`

* Fglrx is the name of AMD's graphics driver for the GPU

Setting up with AMD CPU

- Download the AMD APP SDK from their website
- Extract with `tar -zxf file.tar.gz`
- Install with
  - `sudo ./Install*.sh`
- Create symbolic links to the library and includes
  - `sudo ln -s /opt/AMDAPP/lib/x86_64/* /usr/local/lib`
  - `sudo ln -s /opt/AMDAPP/include/* /usr/local/include`
- Update linker paths
  - `sudo ldconfig`
- Reboot and run `clinfo`
  - Your CPU should be listed

Setting up with AMD APU

- The easiest way is to follow the AMD GPU instructions to install fglrx.
- This means you can use the CPU and GPU parts of your APU as separate OpenCL devices.
- You may have to force your BIOS to use integrated graphics if you have a dedicated GPU too.

Setting up with Intel CPU

- requires an Intel® Xeon™ processor on Linux
- Download the Xeon Linux SDK from the Intel website
- Extract the download
  - `tar -zxf download.tar.gz`
- Install some dependancies
  - `sudo apt-get install rpm alien libnuma1`
- Install them using alien
  - `sudo alien –i *base*.rpm *intel-cpu*.rpm *devel*.rpm`
- Copy the ICD to the right location
  - `sudo cp /opt/intel/<version>/etc/intel64.icd /etc/OpenCL/vendors/`

Setting up with Intel GPU

- requires an Intel® Xeon™ processor on OS X
- This works out of the box!
- Just select the Intel® GPU device
- Intel® also have a driver for Windows: https://software.intel.com/en-us/articles/opencl-drivers

Setting up with Intel® Xeon Phi™

- Intel® Xeon Phi™ coprocessor are specialist processor only found in dedicated HPC clusters.
- As such, we expect most users will be using them in a server environment set up by someone else – hence we wont discusses setting up OpenCL on the Intel® Xeon Phi™ coprocessor in these slides

Setting up with NVIDIA GPUs

- Blacklist the open source driver (IMPORTANT)
  - `sudo nano /etc/modprobe.d/blacklist.conf`
  - Add the line: blacklist nouveau
- Install some dependencies
  - `sudo apt-get install build-essential linux-header-generic opencl-headers`
- Download the NVIDIA driver from their website and unpack the download
- In a virtual terminal (Ctrl+Alt+F1), stop the windows manager
  - `sudo service lightdm stop`
- Give the script run permissions then run it
  - `chmod +x *.run`
  - `sudo ./*.run`
- The pre-install test will fail – this is OK!
- Say yes to DKMS, 32-bit GL libraries and to update your X config
- Reboot!

Installing pyopencl

- Make sure you have python installed
- Install the numpy library
  - `sudo apt-get install python-numpy`
- Download the latest version from the pyopencl website
  - Extract the package with `tar -zxf`
- Run to install as a local package
  - `python setup.py install --user`

C/C++ linking (gcc/g++)

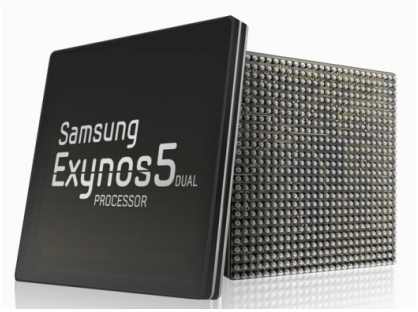
- In order to compile your OpenCL program you must tell the compiler to use the OpenCL library with the flag: `-lOpenCL`
- If the compiler cannot find the OpenCL header files (it should do) you must specify the location of the CL/ folder with the `-I` (capital “i”) flag
- If the linker cannot find the OpenCL runtime libraries (it should do) you must specify the location of the lib file with the `-L` flag
- Make sure you are using a recent enough version of gcc/g++ - at least v4.7 is required to use the OpenCL C++ API (which needs C++11 support)

AN OVERVIEW OF OPENCL

It's a Heterogeneous world

A modern computing platform includes:

- One or more CPUs
- One of more GPUs
- DSP processors
- Accelerators
- … other?



E.g. Samsung® Exynos 5:

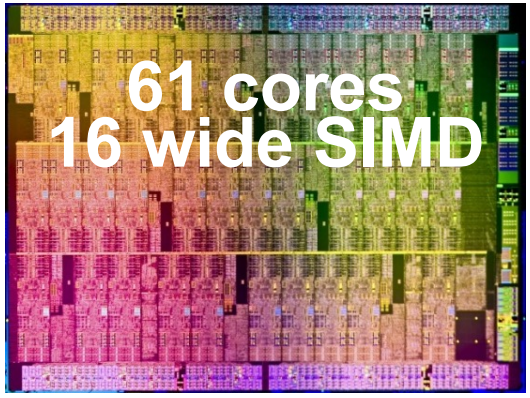
- Dual core ARM A15 1.7GHz, Mali T604 GPU

E.g. Intel XXX with IRIS

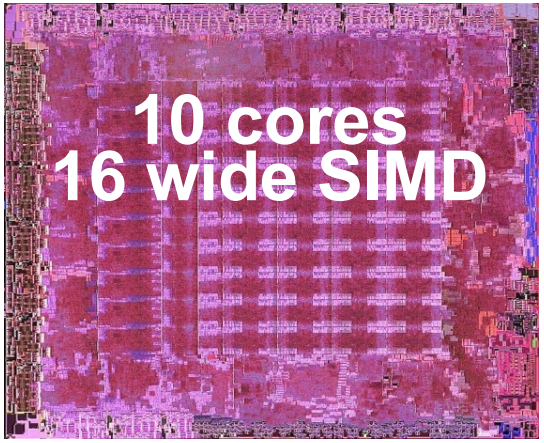
OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform

Microprocessor trends

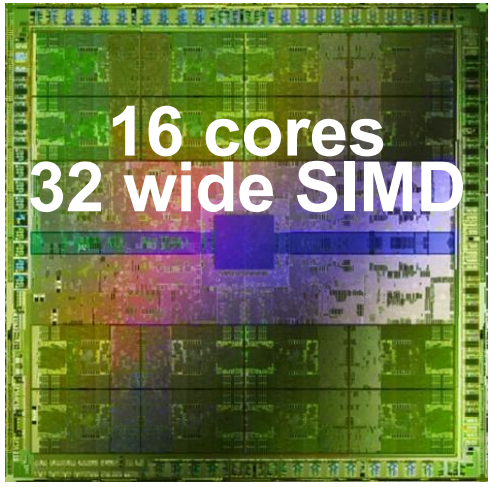
Individual processors have many (possibly heterogeneous) cores.



Intel® Xeon Phi™ coprocessor



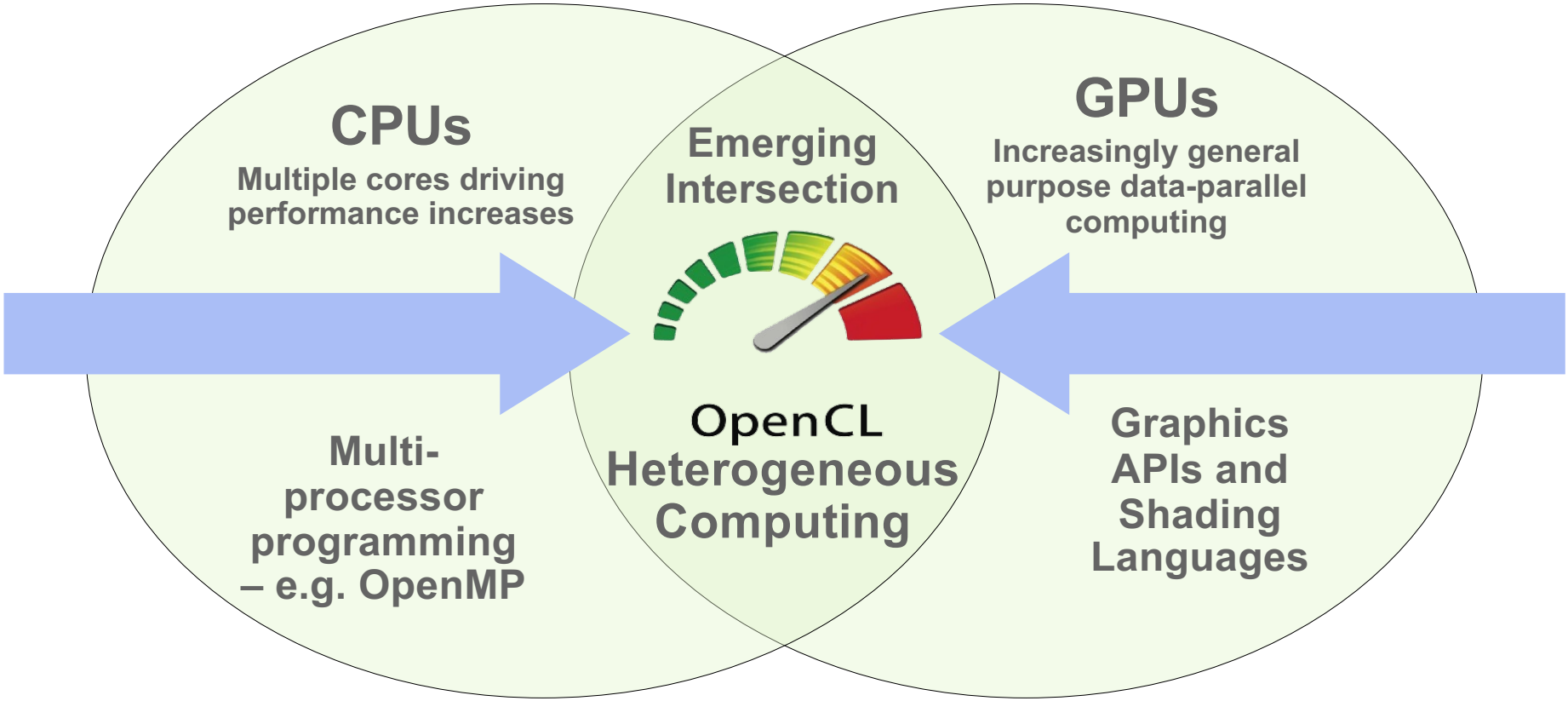
ATI™ RV770



NVIDIA® Tesla® C2090

The Heterogeneous many-core challenge:

How are we to build a software ecosystem for the Heterogeneous many core platform?



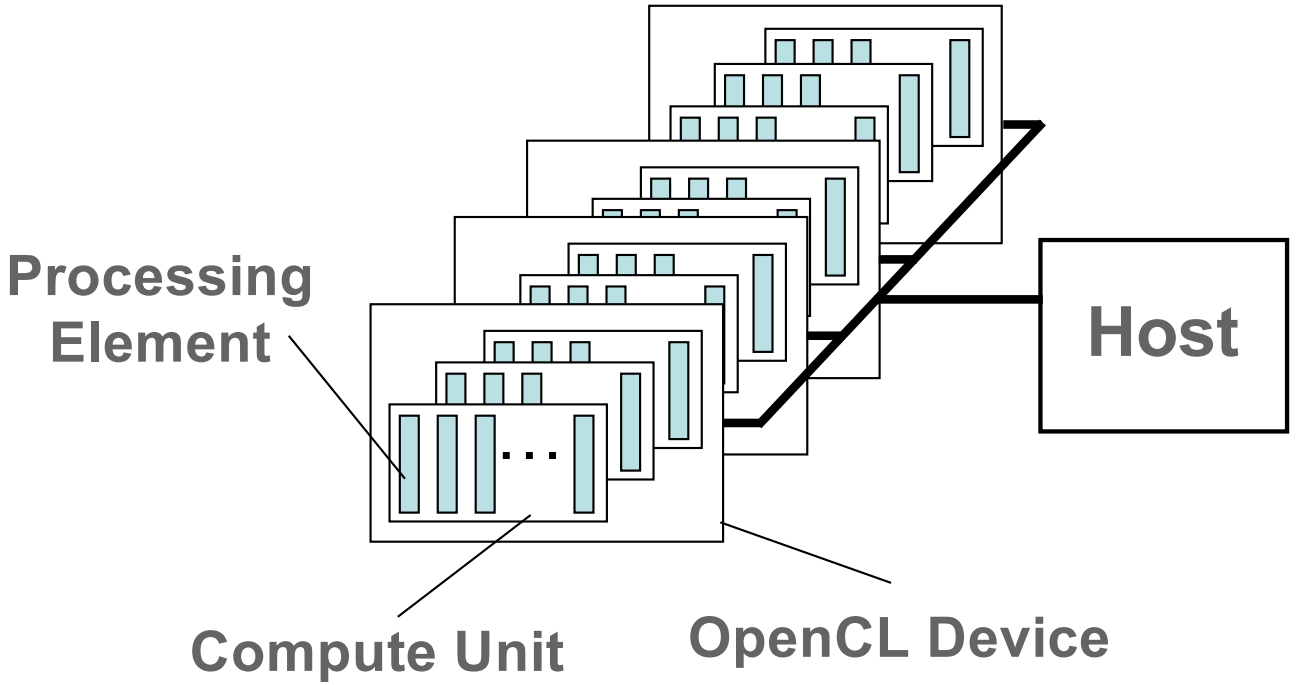
OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

OpenCL Working Group within Khronos

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers.
- OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.

OpenCL Platform Model



- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
    - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

OpenCL Platform Example

CPUs:

- Treated as one OpenCL device
  - One CU per core
  - 1 PE per CU, or if PEs mapped to SIMD lanes, *n* PEs per CU, where *n* matches the SIMD width
  - CPU is its own host!

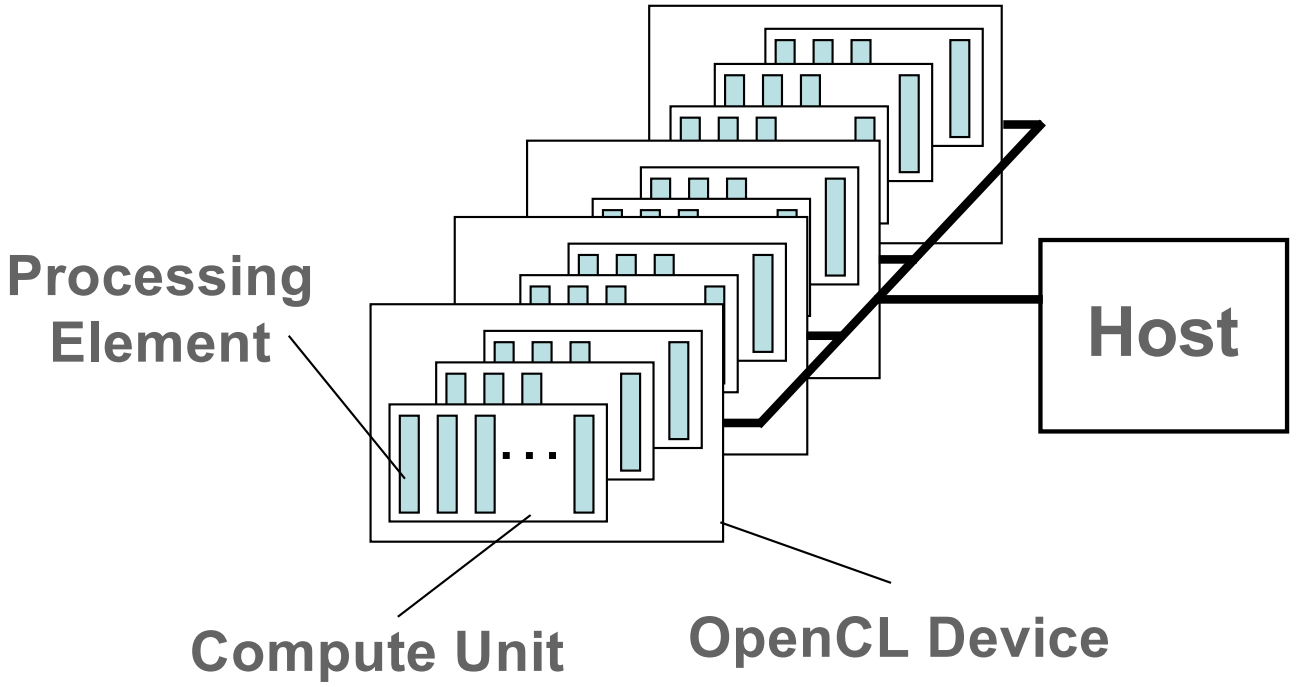
GPUs:

- Each GPU is a separate OpenCL device
- Can use CPU and all GPU devices concurrently through OpenCL

CU = Compute Unit; PE = Processing Element
SIMD = Single Instruction, Multiple Data

IMPORTANT OPENCL CONCEPTS

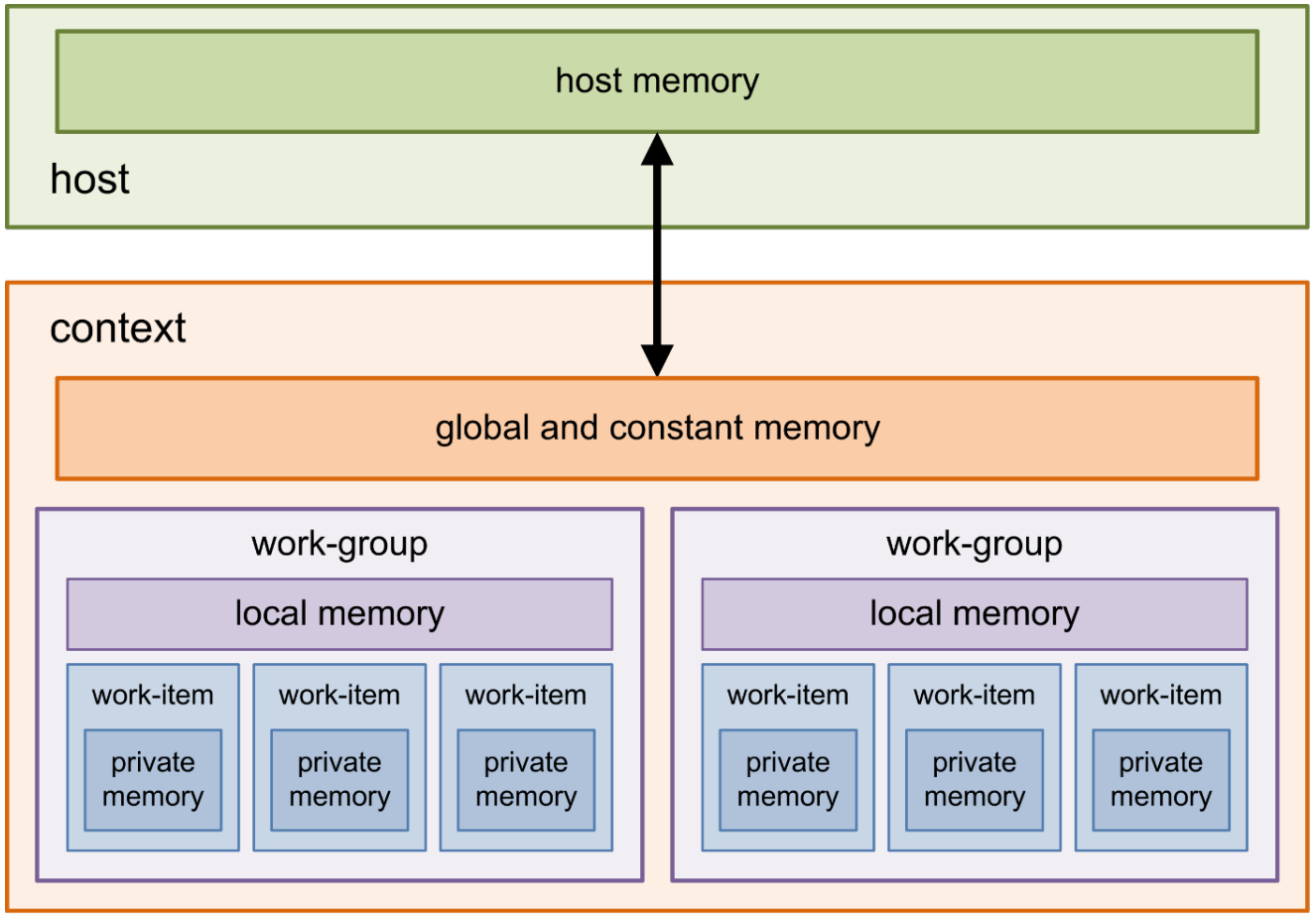
OpenCL Platform Model



- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
    - Each Compute Unit is divided into one or more ***Processing Elements***

OpenCL platform

- Memory divided into ***host memory*** and ***device memory***



The idea behind OpenCL

- Replace loops with functions (a kernel) executing at each point in a problem domain
  - E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

Traditional loops

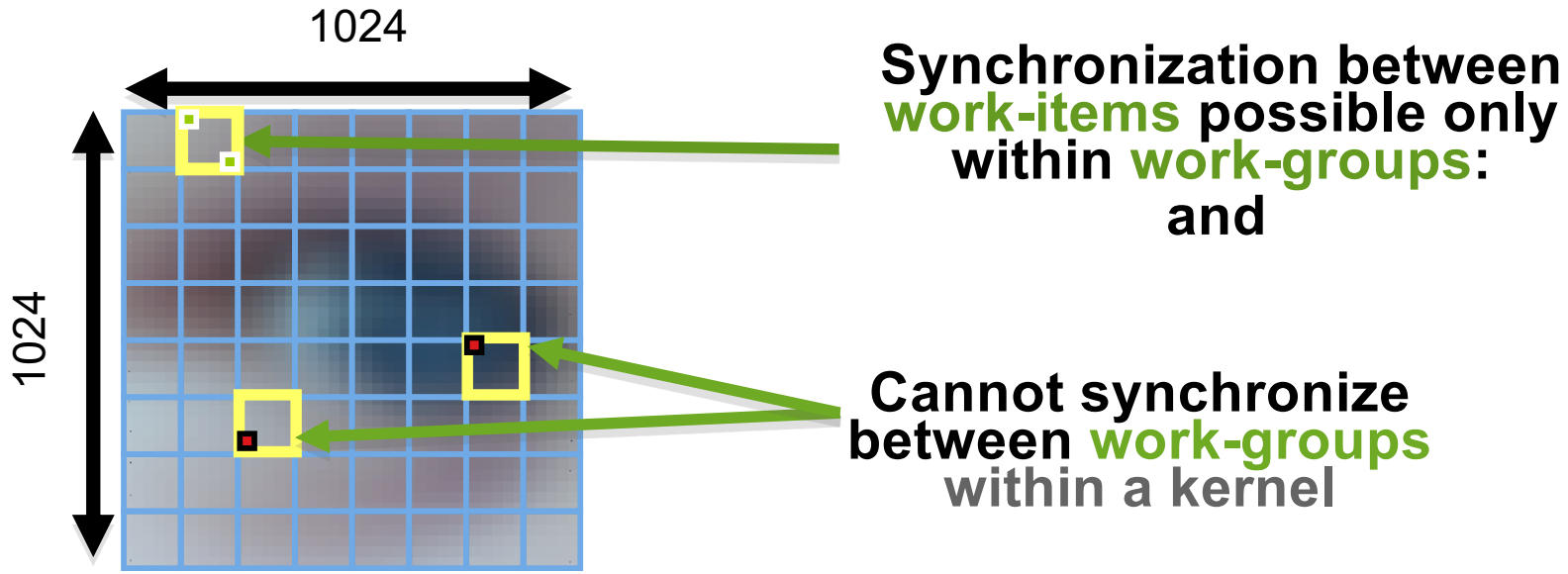
```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
// many instances of the kernel,
// called work-items, execute
// in parallel
```

An N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 64x64 (**work-group**, executes together)



Choose the dimensions that are “best” for your algorithm

OpenCL N Dimensional Range (NDRange)

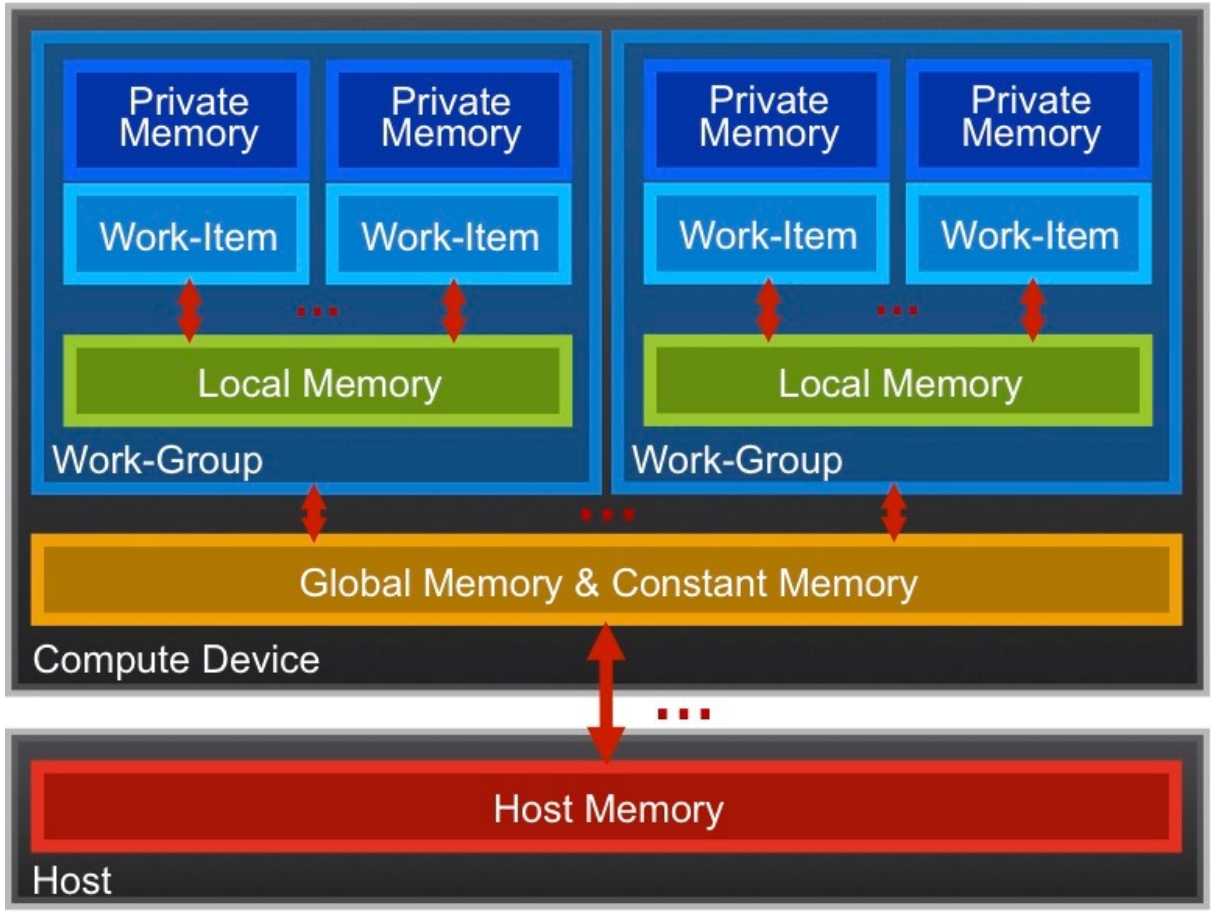
- The problem we want to compute should have some **dimensionality**;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension – this is called the **global** size
- We associate each point in the iteration space with a **work-item**

OpenCL N Dimensional Range (NDRange)

- Work-items are grouped into **work-groups**; work-items within a work-group can share **local memory** and can **synchronize**
- We can specify the number of work-items in a work-group – this is called the **local** (work-group) size
- Or the OpenCL run-time can choose the work-group size for you (usually not optimal)

OpenCL Memory model

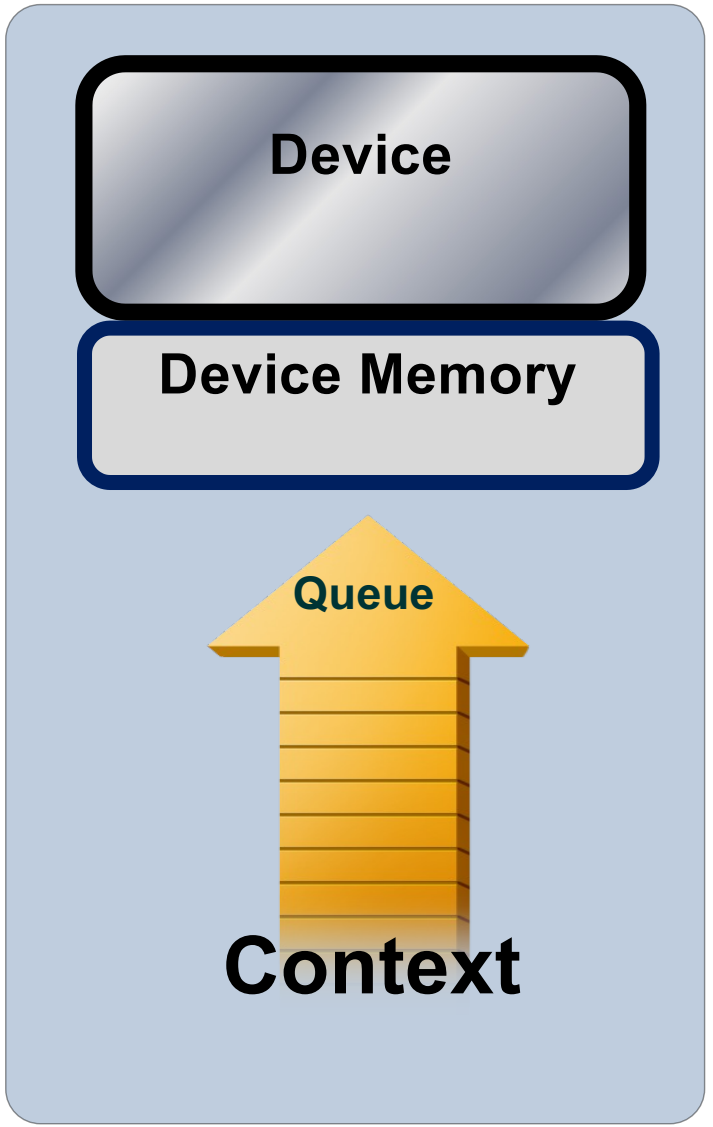
- ***Private Memory***
  - Per work-item
  - Shared within a work-group
- ***Global Memory /Constant Memory***
  - Visible to all work-groups
- ***Host memory***
  - On the CPU



Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

Context and Command-Queues

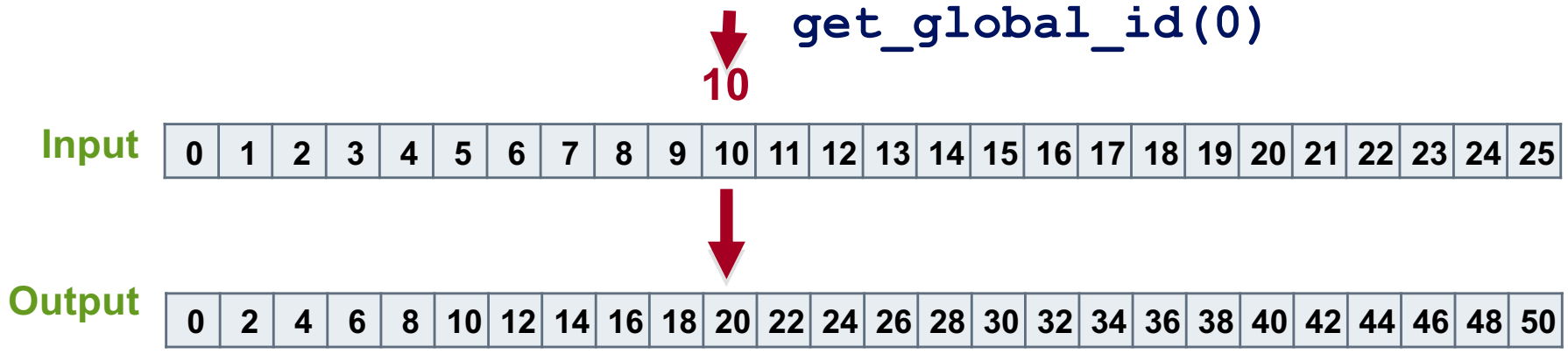
- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a **kernel** for each point in the domain

```
__kernel void times_two(
    __global float* input,
    __global float* output)
{
    int i = get_global_id(0);
    output[i] = 2.0f * input[i];
}
```

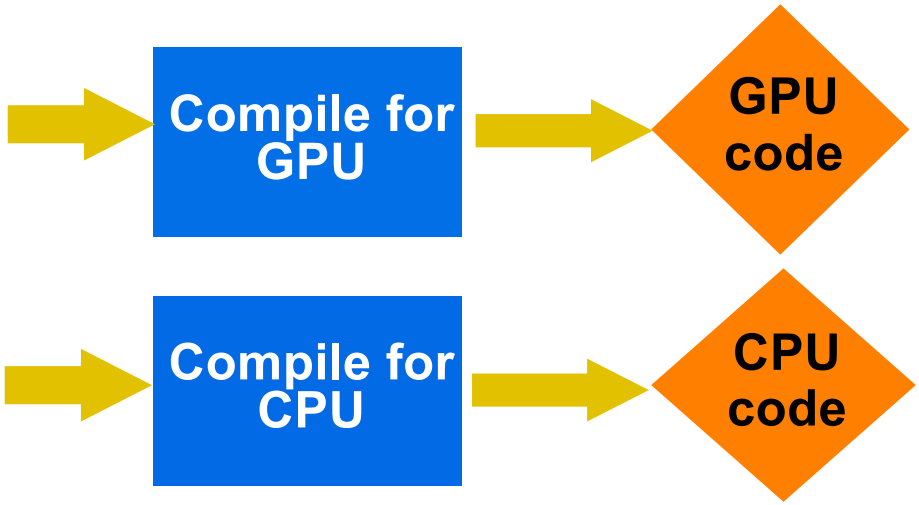


Building Program Objects

- The program object encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options
- The C API build a process to create a program object:
  - `clCreateProgramWithSource()`
  - `clCreateProgramWithBinary()`

OpenCL uses runtime compilation … because in general you don't know the details of the target device when you ship the program

```
__kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                               (int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



Example: vector addition

- The “hello world” program of data parallel programming is a program to add two vectors

```
C[i] = A[i] + B[i] for i=0 to N-1
```

- For the OpenCL solution, there are two parts
  - Kernel code
  - Host code

Vector Addition - Kernel

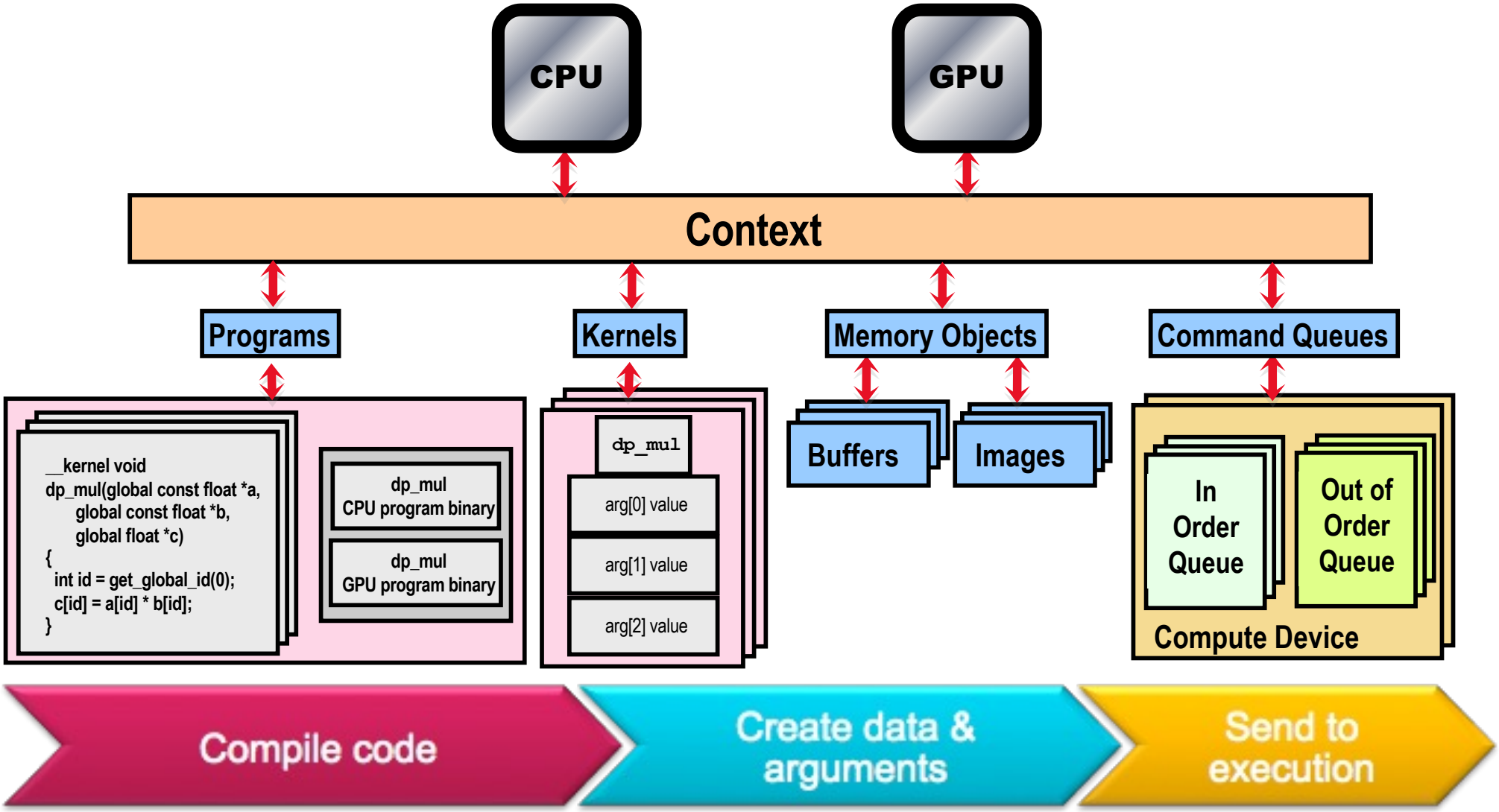
```
__kernel void vadd(__global const float *a,
                   __global const float *b,
                   __global       float *c)
{
    int gid = get_global_id(0);
    c[gid]  = a[gid] + b[gid];
}
```

Vector Addition – Host

- The host program is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels
- 5 simple steps in a basic host program:
  1. Define the ***platform*** … platform = devices+context+queues
  2. Create and Build the ***program*** (dynamic library for kernels)
  3. Setup ***memory*** objects
  4. Define the ***kernel*** (attach arguments to kernel functions)
  5. Submit ***commands*** … transfer memory objects and execute kernels

See reference card at OpenCL:
https://www.khronos.org/files/opencl-quick-reference-card.pdf

▸The basic platform and runtime APIs in OpenCL (using C)



1. Define the platform

- Grab the first available platform:

```
err = clGetPlatformIDs(1, &firstPlatformId,
                                    &numPlatforms);
```

- Use the first CPU device the platform provides:

```
err = clGetDeviceIDs(firstPlatformId,
                CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
```

- Create a simple context with a single device:

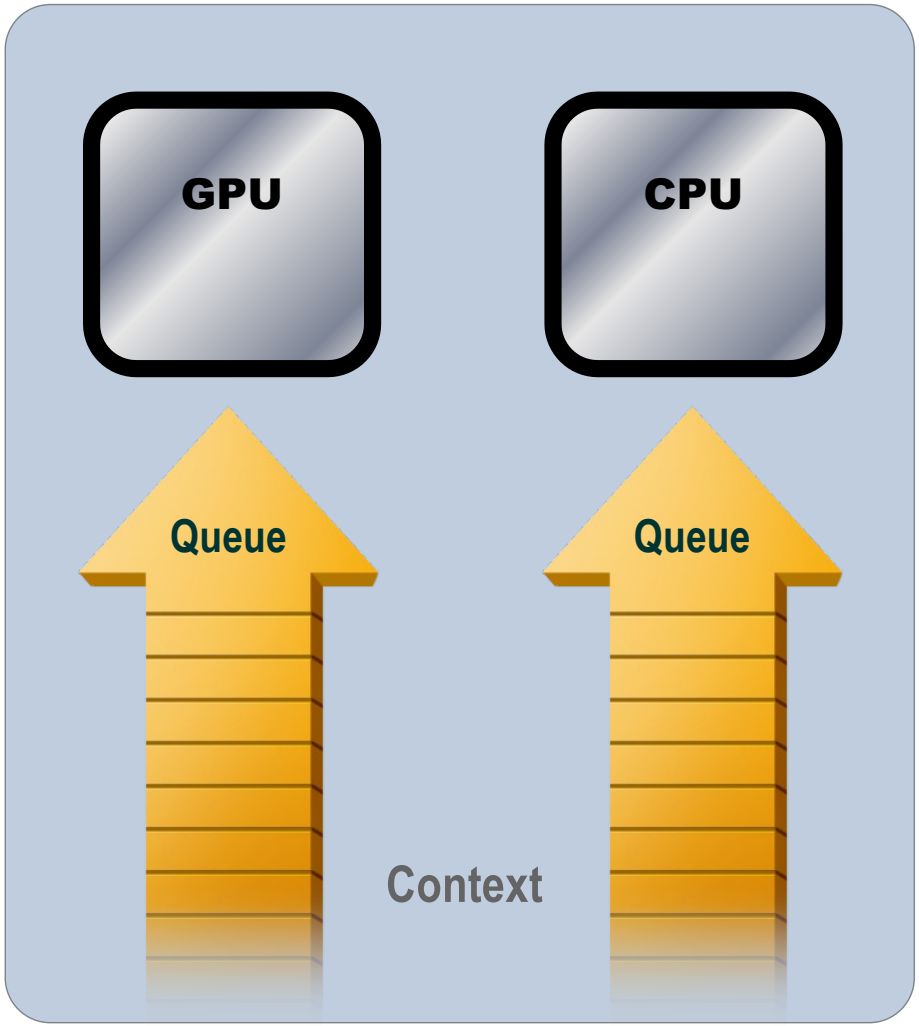
```
context = clCreateContext(firstPlatformId, 1,
                                &device_id, NULL, NULL, &err);
```

- Create a simple command-queue to feed our device:

```
commands = clCreateCommandQueue(context, device_id,
                                                0, &err);
```

Command-Queues

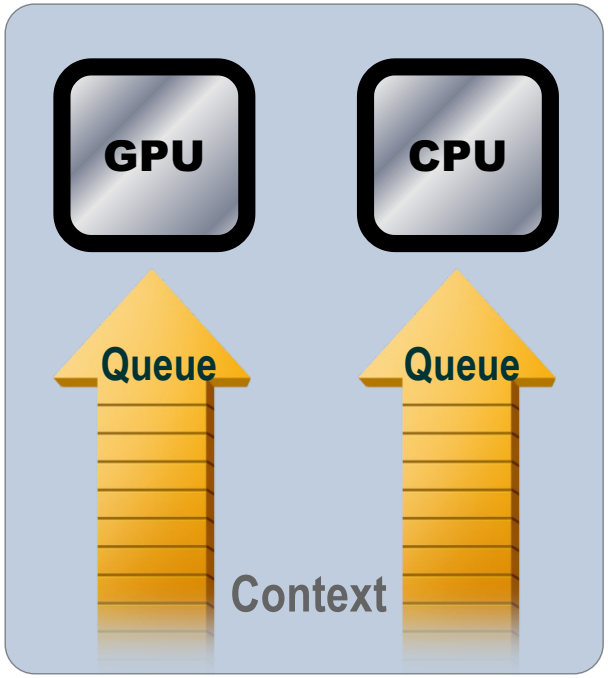
- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define independent streams of commands that don't require synchronization



Command-Queue execution details

Command queues can be configured in different ways to control how commands execute

- ***In-order queues***:
  - Commands are enqueued and complete in the order they appear in the program (program-order)
- ***Out-of-order queues***:
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.
- Execution of commands in the command-queue are guaranteed to be completed at synchronization points



2. Create and Build the program

- Define source code for the kernel-program as a string literal or read from a file (for real applications).

- Build the program object:

```
program = clCreateProgramWithSource(context, 1
            (const char**) &KernelSource, NULL, &err);
```

- Compile the program to create a “dynamic library” from which specific kernels can be pulled:

```
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);
```

Error messages

- Fetch and print error messages:

```
if (err != CL_SUCCESS) {
 size_t len;
 char buffer[2048];
 clGetProgramBuildInfo(program, device_id,
  CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
 printf("%s\n", buffer);
}
```

- Important to do check all your OpenCL API error messages!

- Easier in C++ with try/catch (see later)

3. Setup Memory Objects

- For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C.
- Create input vectors and assign values on the host:

```
float h_a[LENGTH], h_b[LENGTH], h_c[LENGTH];
for (i = 0; i < length; i++) {
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
}
```

- Define OpenCL memory objects:

```
d_a = clCreateBuffer(context, CL_MEM_READ_ONLY,
                     sizeof(float)*count, NULL, NULL);
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY,
                     sizeof(float)*count, NULL, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                     sizeof(float)*count, NULL, NULL);
```

What do we put in device memory?

Memory Objects:

- A handle to a reference-counted region of global memory.

There are two kinds of memory object

- ***Buffer*** object:
  - Defines a linear collection of bytes (“*just a C array*”).
  - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
- ***Image*** object:
  - Defines a two- or three-dimensional region of memory.
  - Image data can only be accessed with read and write functions, i.e. these are opaque data structures. The read functions use a sampler.

Used when interfacing with a graphics API such as OpenGL. We won’t use image objects in this tutorial.

Creating and manipulating buffers

- Buffers are declared on the host as type: `cl_mem`
  - Arrays in host memory hold your original host-side data:

```
float h_a[LENGTH], h_b[LENGTH];
```

- Create the buffer (d_a), assign sizeof(float)*count bytes from “h_a” to the buffer and copy it into device memory:

```
cl_mem d_a = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
     sizeof(float)*count, h_a, NULL);
```

Conventions for naming buffers

- It can get confusing about whether a host variable is just a regular C array or an OpenCL buffer

- A useful convention is to prefix the names of your regular **h**ost C arrays with "**h_**" and your OpenCL buffers which will live on the **d**evice with "**d_**"

Creating and manipulating buffers

- Other common memory flags include:

  CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE

- These are from the point of view of the **device**

- Submit command to copy the buffer back to host memory at “h_c”:
  - CL_TRUE = blocking, CL_FALSE = non-blocking

```
clEnqueueReadBuffer(queue, d_c, CL_TRUE,
        sizeof(float)*count, h_c,
        NULL, NULL, NULL);
```

4. Define the kernel

- Create kernel object from the kernel function “vadd”:

```
kernel = clCreateKernel(program, “vadd”, &err);
```

- Attach arguments of the kernel function “vadd” to memory objects:

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int),
                      &count);
```

5. Enqueue commands

- Write Buffers from host into global memory (as non-blocking operations):

```
err = clEnqueueWriteBuffer(commands, d_a, CL_FALSE,
          0, sizeof(float)*count, h_a, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, d_b, CL_FALSE,
          0, sizeof(float)*count, h_b, 0, NULL, NULL
```

- Enqueue the kernel for execution:

```
err = clEnqueueNDRangeKernel(commands, kernel, 1,
                  NULL, &global, &local, 0, NULL, NULL);
```

5. Enqueue commands

- Read back result (as a blocking operation). We have an in-order queue which assures the previous commands are completed before the read can begin.

```
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE,
                sizeof(float)*count, h_c, 0, NULL, NULL);
```

Vector Addition – Host Program

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
                     CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
                                &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                            sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
                            sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                            sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                    global_work_size, NULL,0,NULL,NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                          CL_TRUE, 0,
                          n*sizeof(cl_float), dst,
                          0, NULL, NULL);
```

Vector Addition – Host Program

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
                     CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetCo                                              cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

// allocate the buffer memory objects
memobjs[0                                                    |
       CL                                                    NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);

// create th
program = cl
                                  &program_source, NULL, NULL);
```

Define platform and queues

Define memory objects

Create the program

```
// build the pr
err = clBuildPr                                    ,NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],

err |= clS                                         ],
                                   sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                                   sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueu                                     1, NULL,
                                                   L,NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                                                   0, NULL, NULL);
```

Build the program

Create and setup kernel

Execute the kernel

Read results on the host

It's complicated, but most of this is "boilerplate" and not as bad as it looks.

Exercise 1

- Implement an OpenCL program that prints out the information of your platform (computer)

Exercise 2

- Implement the vector addition example with 2 vectors
- Implement the vector addition example with 3 vectors