CUDA PROGRAMMING AND TOOLS

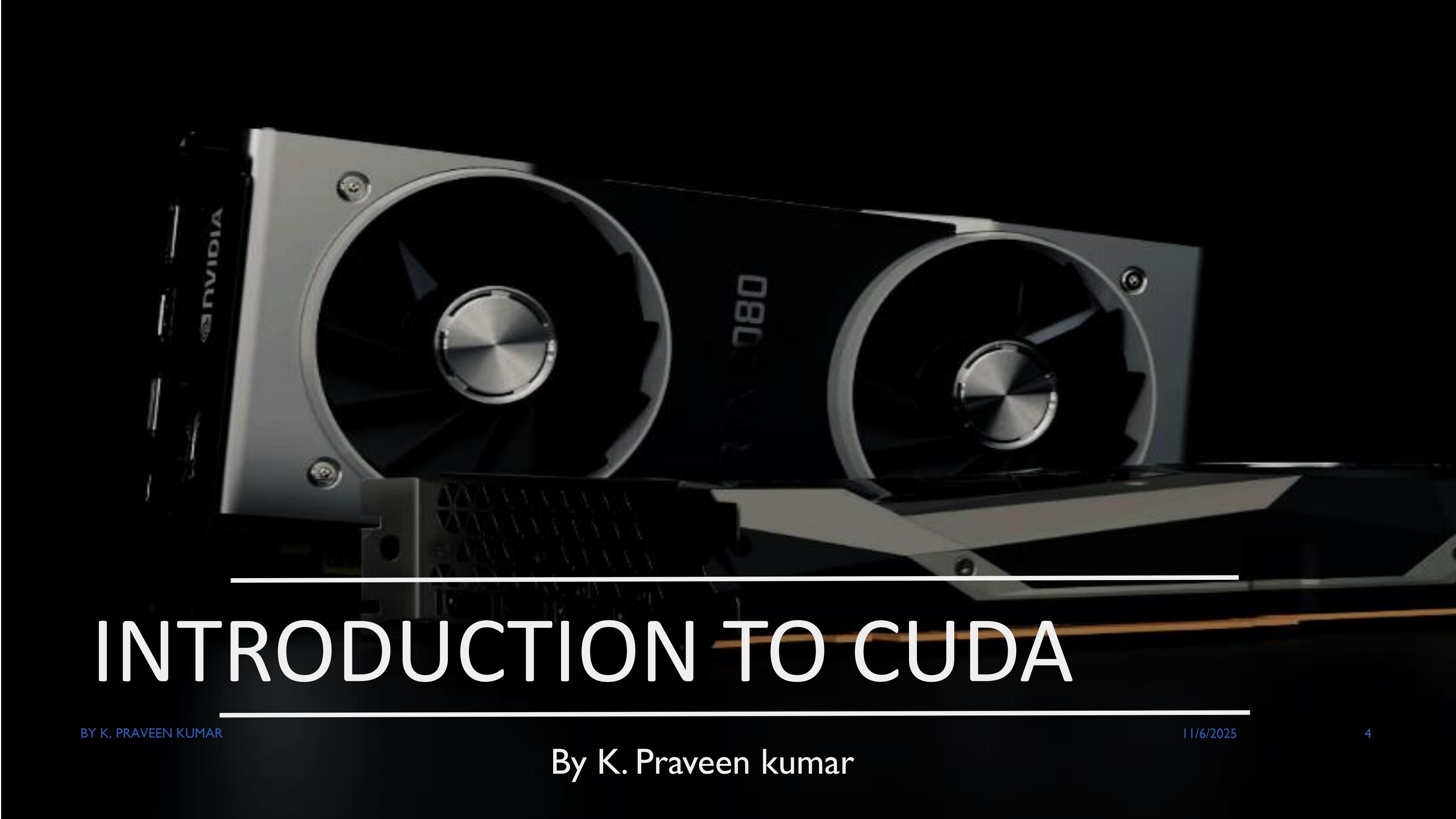
BY K. PRAVEEN KUMAR

MY INTRODUCTION

- Name :- K. Praveen Kumar
- linkedIn :- https://www.linkedin.com/in/praveenkumarkrishnaiah/

EXPERIENCE SUMMARY
• 25+ years of work experience on Telecom, Healthcare, Directory Services domains and corporate trainer and consultant, and Entrepreneur. • Technical consultant for startups and Freelance Corporate Trainer. • Technical Consultant of Techance Software Pvt. Ltd. • Senior Systems Specialist & Lead in GE Medical Systems (I) Pvt. Ltd. • Senior Software Engineer, Scrum master & Team Lead in Nokia Software (I) Pvt. Ltd. • Technical Consultant Engineer in Colossal Technologies Pvt Ltd. • Free-lancing as a corporate trainer in various technologies • Software Engineer in Novell Software Development (I) Ltd., Bangalore • Software Engineer in Synergy Infotech Pvt. Ltd., Bangalore.

CONTENTS



INTRODUCTION TO CUDA

By K. Praveen kumar

WHAT IS CUDA?

- CUDA is NVIDIA’s parallel computing platform that lets developers run many tasks simultaneously on GPUs.
- It provides extensions to C/C++ so programmers can use GPUs for faster general-purpose computing.
- CUDA unlocks the power of thousands of GPU cores to accelerate complex calculations beyond graphics rendering.

NVIDIA®

But Why GPUs for Computation?

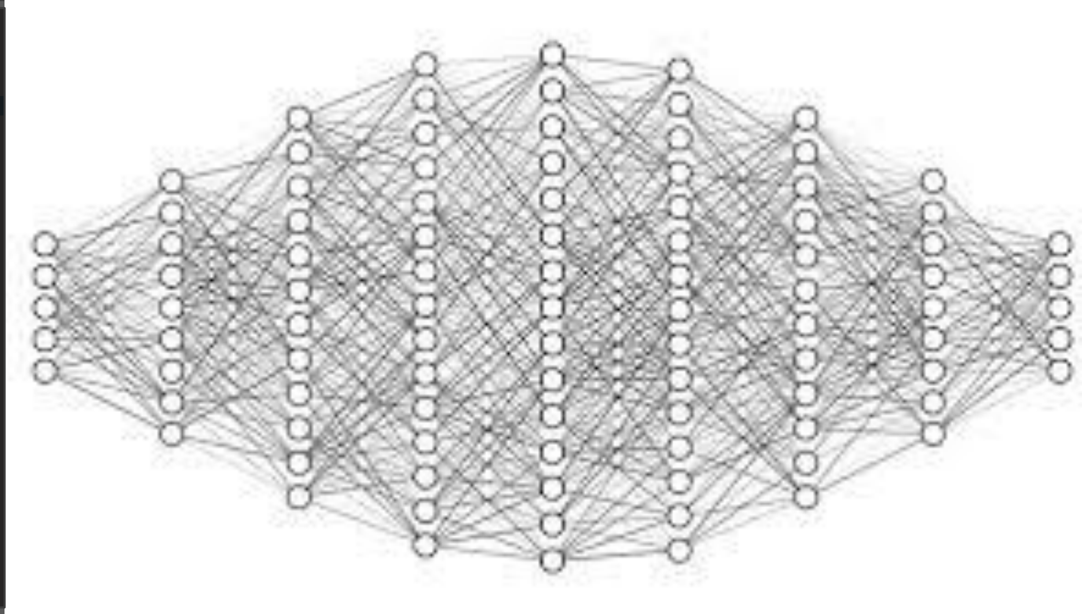
GPUs have thousands of simple cores for running many tasks at once.

GPUs excel at processing large data sets in parallel.

Parallelism in GPUs speeds up tasks like AI and graphics.

CPUs have fewer, more powerful cores for sequential tasks.

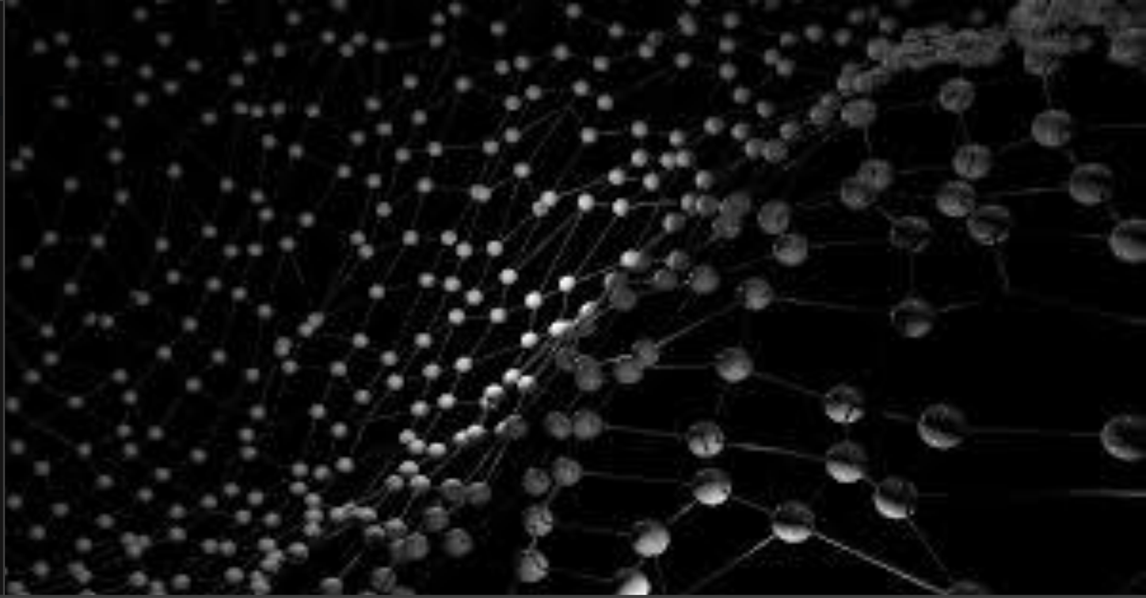
Researchers use GPUs to train AI models quickly and efficiently. These accelerated computations make deep learning and data science much more practical for real-world problems.



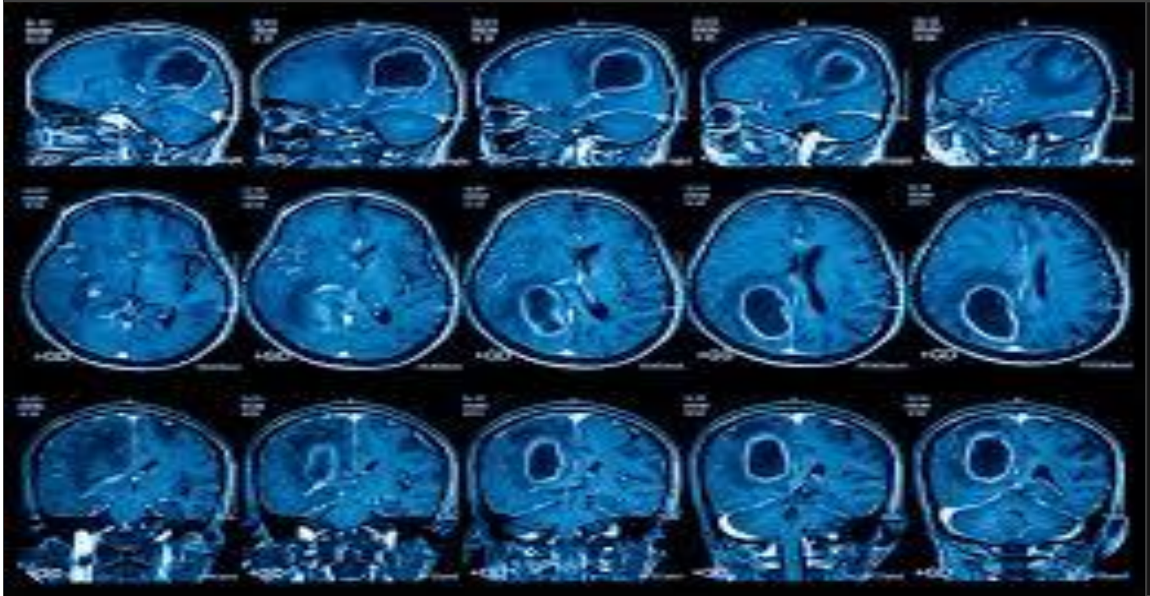
REAL WORLD APPLICATIONS OF GPUs

Modern video games leverage GPU power to render lifelike graphics in real time. This enables immersive experiences and supports new technologies like virtual reality.

Scientists rely on GPUs for molecular modeling and running weather simulations faster than ever. GPU acceleration allows them to analyze complex systems with greater accuracy.

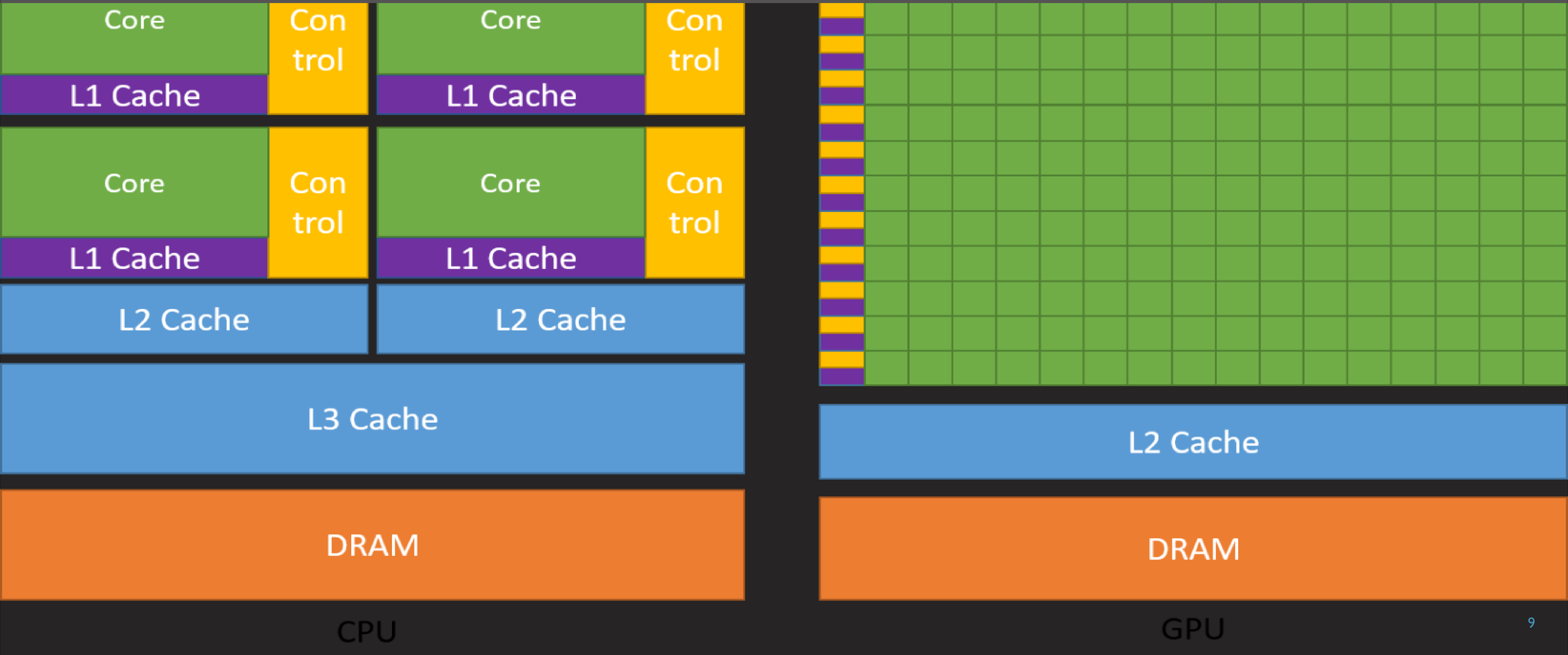


REAL WORLD APPLICATIONS OF GPUs

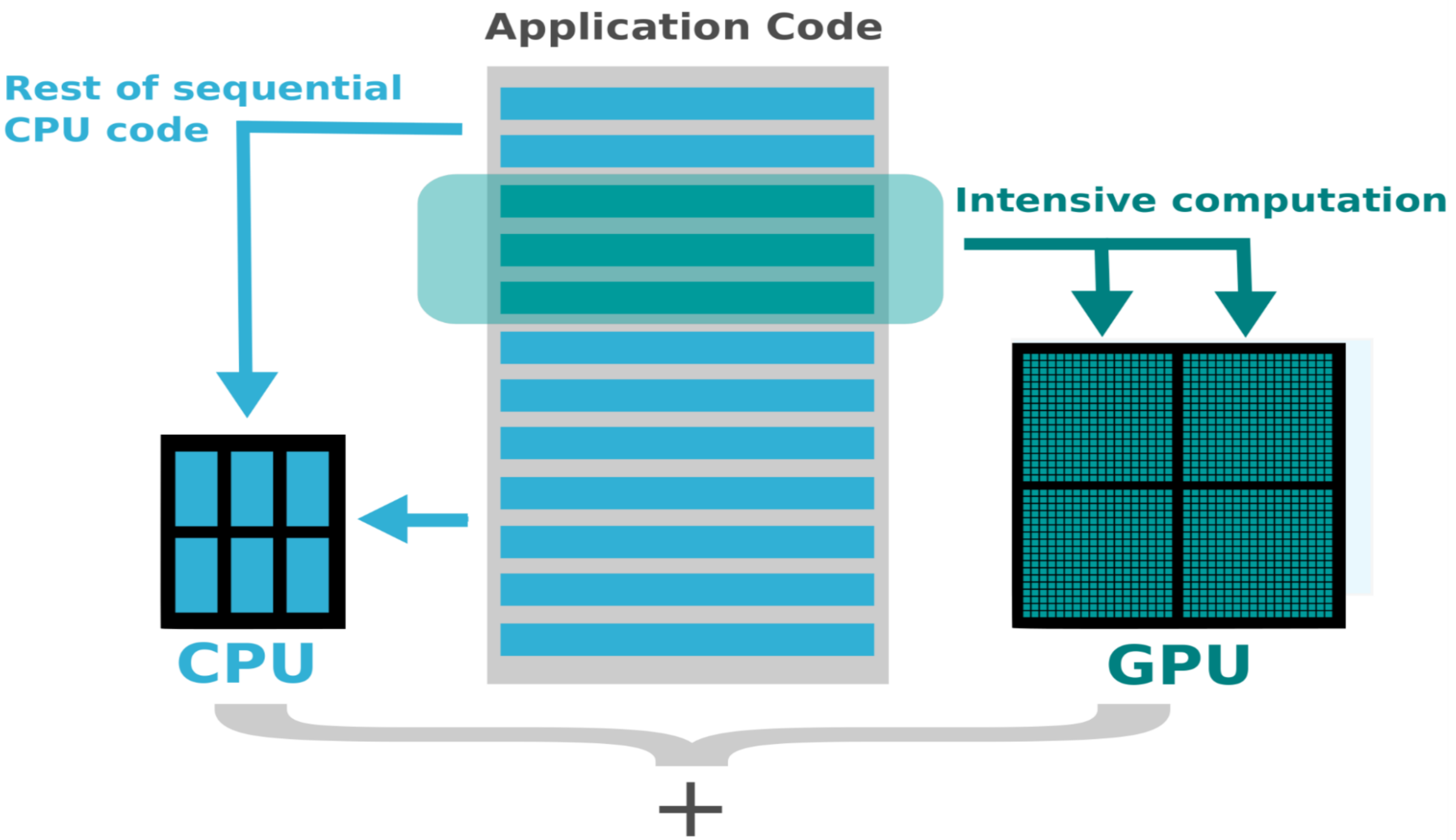


Doctors process and analyze medical images using GPUs to detect health issues more rapidly. Fast GPU performance means quicker diagnoses and better treatment decisions.

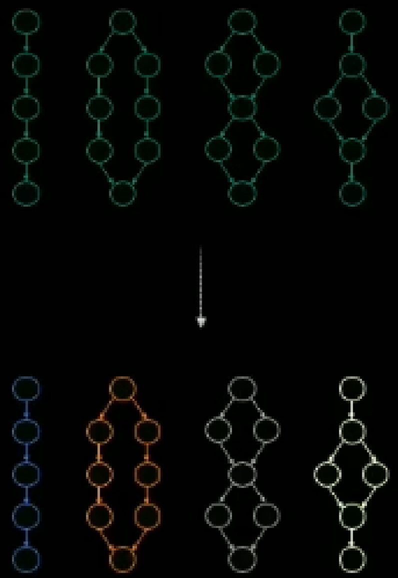
CPU vs GPU ARCHITECTURE



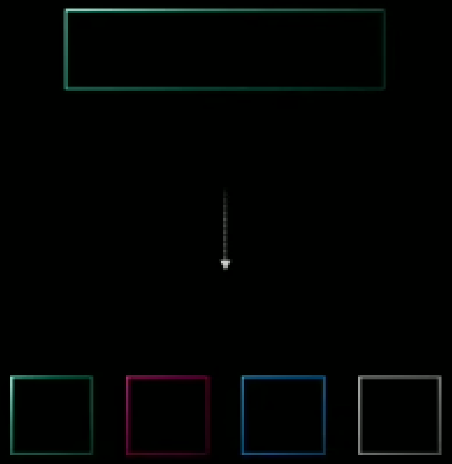
HOW GPU ACCELERATION ACTUALLY WORKS?



There Are Really Only Two Types of Parallelism Patterns



Task Parallelism
Divide independent **programs** across processors



Data Parallelism
Divide individual **data elements** across processors

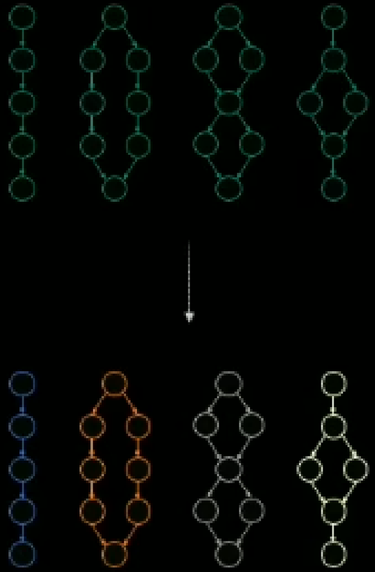
Task Parallelism

- Independent programs running on different threads/processors at the same time
- Could be copies of the same program at different positions in the code

Data Parallelism

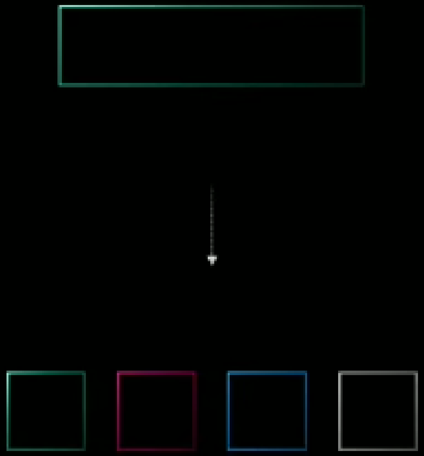
- A single program running across multiple threads/processors
- All threads execute the same operation on different elements of data in lockstep

CUDA is Both: Data Parallelism Inside Task Parallelism



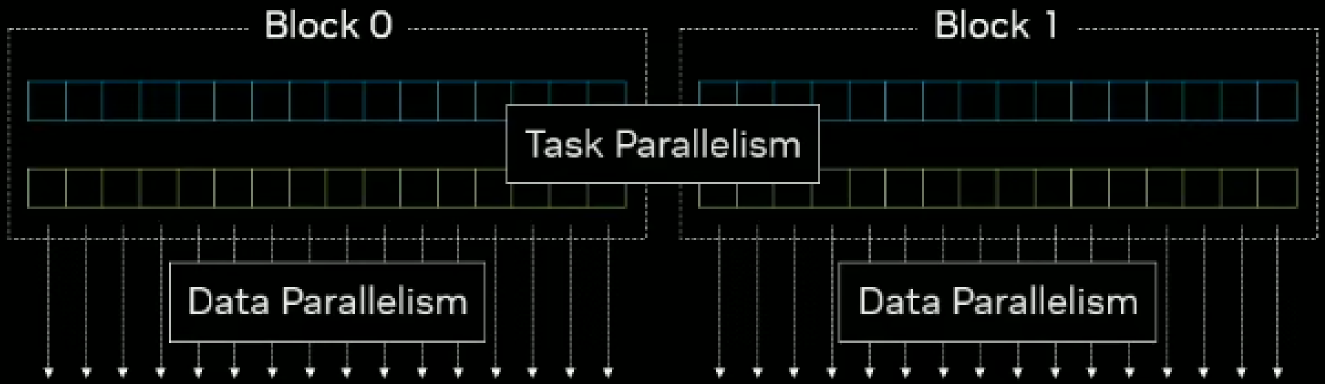
Task Parallelism

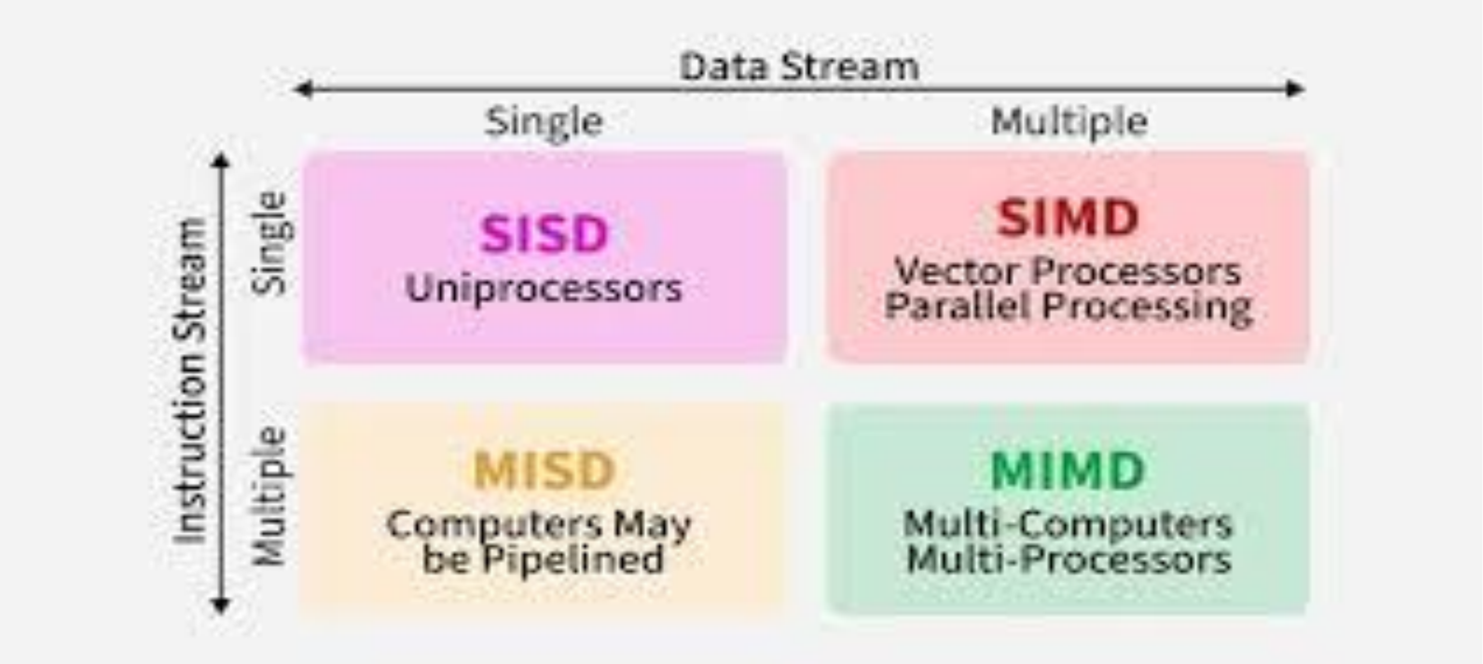
Divide independent **programs** across processors

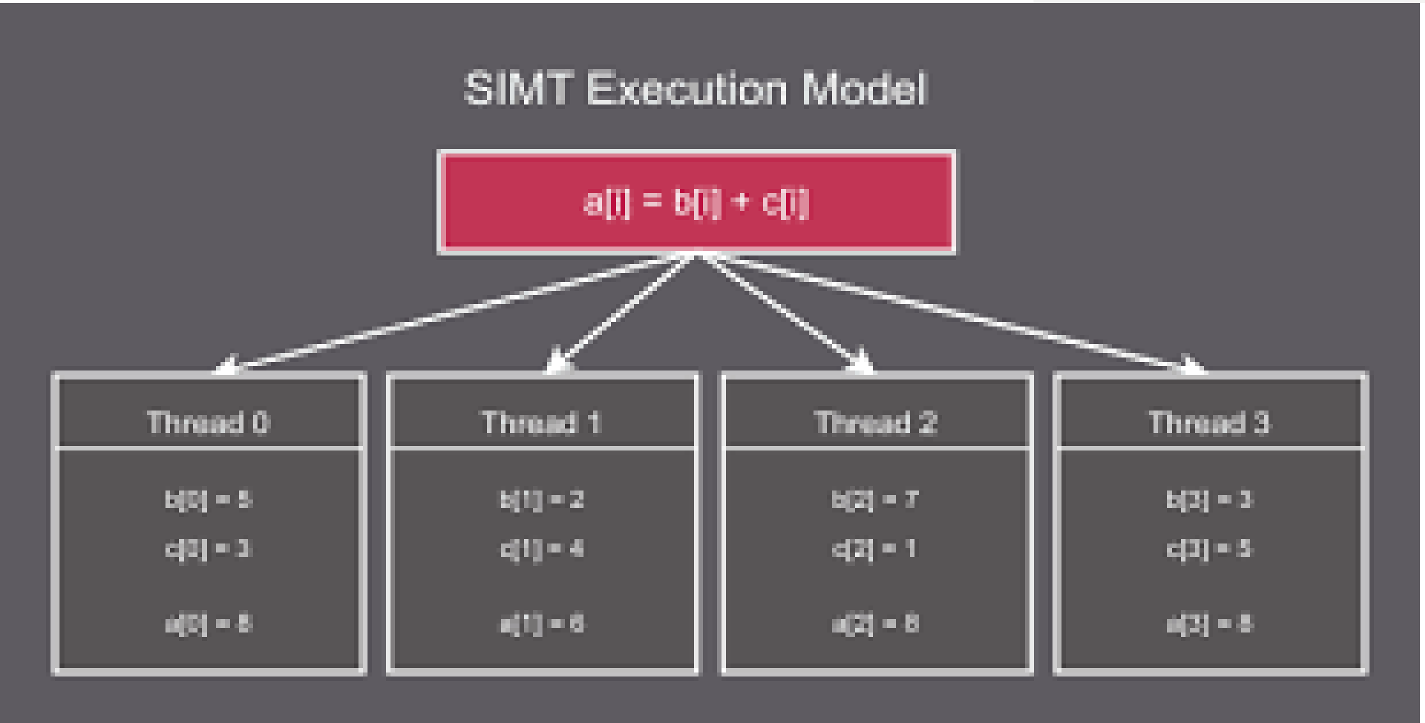


Data Parallelism

Divide individual **data elements** across processors




Data Stream
Single
Multiple
Instruction Stream
Single
Multiple
SISD
Uniprocessors
SIMD
Vector Processors
Parallel Processing
MISD
Computers May
be Pipelined
MIMD
Multi-Computers
Multi-Processors

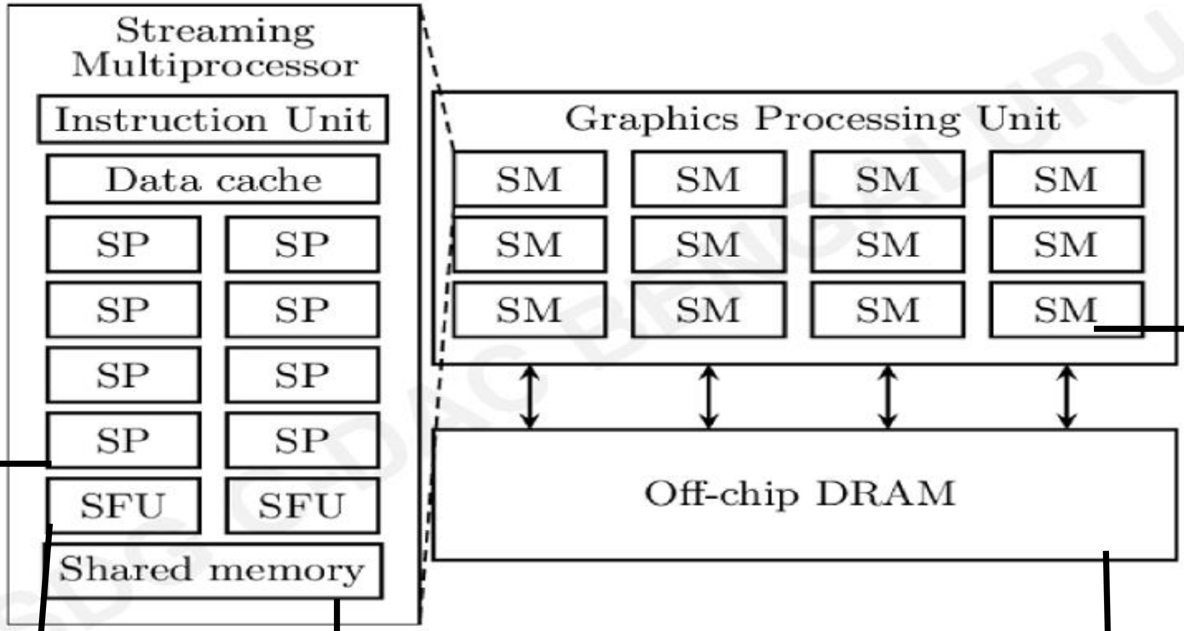

SIMT Execution Model
a[i] = b[i] + c[i]
Thread 0
b[0] = 5
c[0] = 3
a[0] = 8
Thread 1
b[1] = 2
c[1] = 4
a[1] = 6
Thread 2
b[2] = 7
c[2] = 1
a[2] = 8
Thread 3
b[3] = 3
c[3] = 5
a[3] = 8

Streaming Processors

-Scalar Processor within a SM that executes floating point operations for a single thread.

-Also known as CUDA core.

-Analogous to ALU in CPUs



Streaming Multiprocessor

-executes multiple threads in parallel

-Uses SIMT model

-contains multiple SPs, SFUs, register files, shared memory etc.

Special Function Unit used to perform transcendental operations such as sine, cosine, reciprocal, square root, etc.

-Shared among threads of same thread block.

-Faster than Global memory but limited in size(~48-100 KB per SM)

-Main memory (Global memory) of the GPU.

-Accessible by all threads.

-Requires explicit copy from host to device

CUDA Environment Setup

CUDA development requires CUDA-enabled NVIDIA GPU with at least 256MB of memory.

- cmd> lspci | grep -i nvidia

Latest NVIDIA drivers must be installed to ensure compatibility with CUDA.

- cmd> nvidia-smi

Must have CUDA Toolkit provides the compiler (nvcc), libraries, and tools needed to build CUDA applications.

- cmd> nvcc –version

A compatible host compiler is required: GCC on Linux.

CUDA Environment Setup

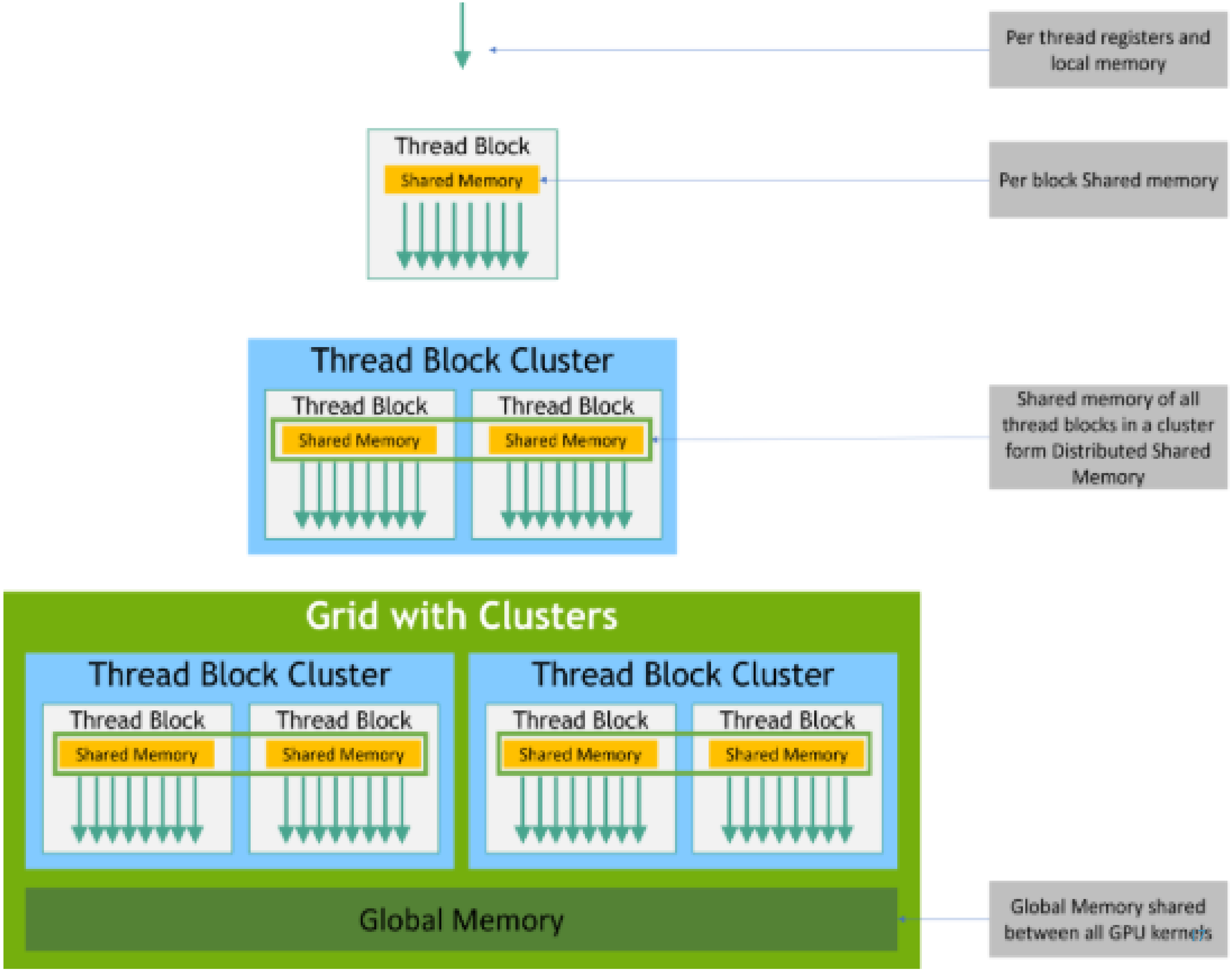
CUDA development requires two compilers: one for the CPU (host) code and one for the GPU (device) code.

To compile CUDA code:

- cmd> nvcc demo.cu -o demo

To Run CUDA code:

- cmd> ./demo



Per thread registers and local memory
Thread Block
Shared Memory
Per block Shared memory
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Shared memory of all thread blocks in a cluster form Distributed Shared Memory
Grid with Clusters
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Global Memory
Global Memory shared between all GPU kernels

CUDA Execution Hierarchy

Thread :

-Smallest unit of execution in CUDA.
-Executes one instance of kernel.
-Has private registers and local memory.
-Identified by threadIdx.

Warp:

-A group of 32 threads.
-The basic scheduling unit on the GPU.
-All threads in a warp execute the same instruction, but on different data.
-Occupancy: Ratio of active warps to maximum possible warps
-warp divergence: Threads in the same warp execute different paths serially.

CUDA Execution Hierarchy

Block:

-A group of warps.

-Threads in a block can share data via shared memory,

-Identified by blockIdx and defined by blockDim.

-Maximum 1024 threads per block on most CUDA-capable GPUs.

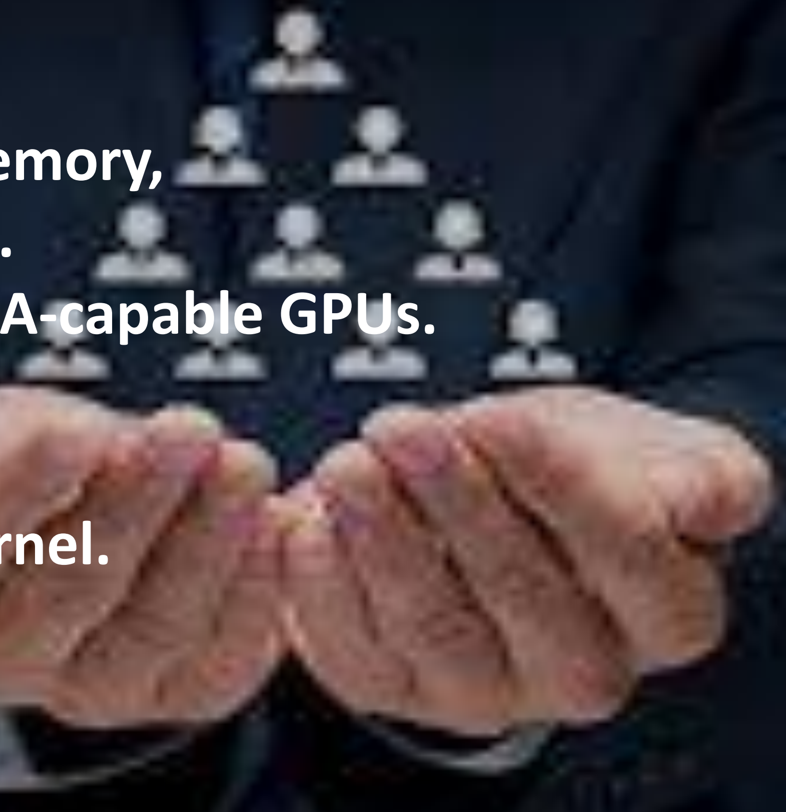
Grid:

-A collection of blocks that execute the same kernel.

-All blocks execute independently.

-Specified during kernel launch using

<<<numBlocks, threadsPerBlock>>>



CUDA Execution Hierarchy

Kernel:
-A GPU function (defined with __global__) launched by the host.
-Kernel can be launched using <<<numBlocks, threadsPerBlock>>>
-If a kernel is launched with more threads than the GPU supports, the kernel fails to launch.

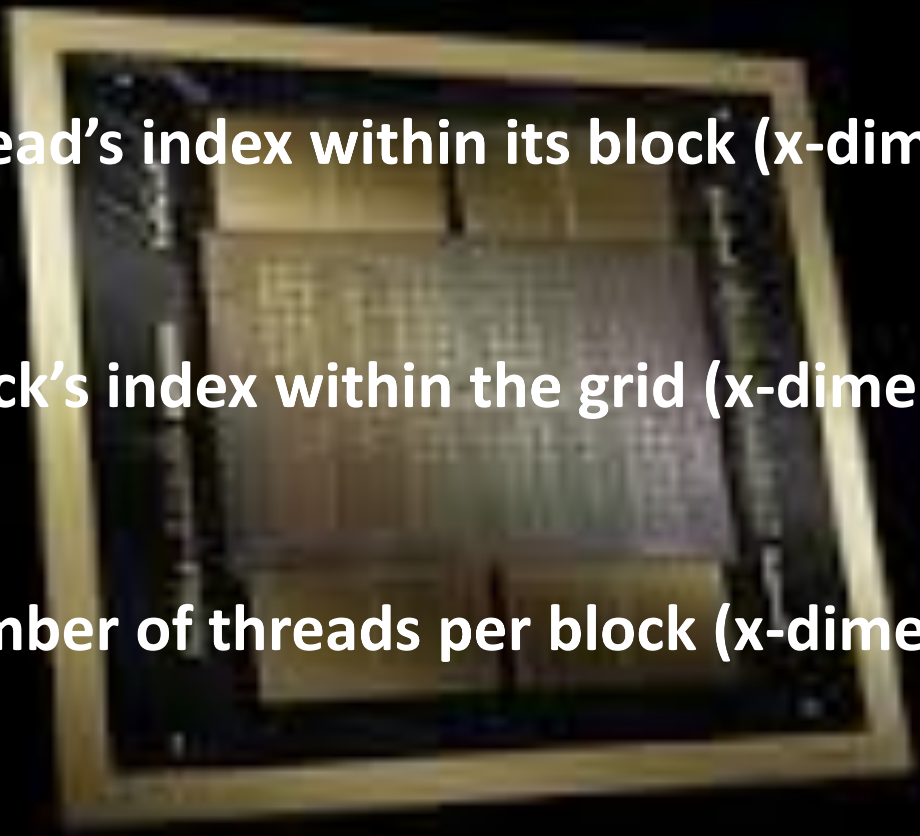
KERNEL > GRID > BLOCK > WARP > THREAD

CUDA Constructs

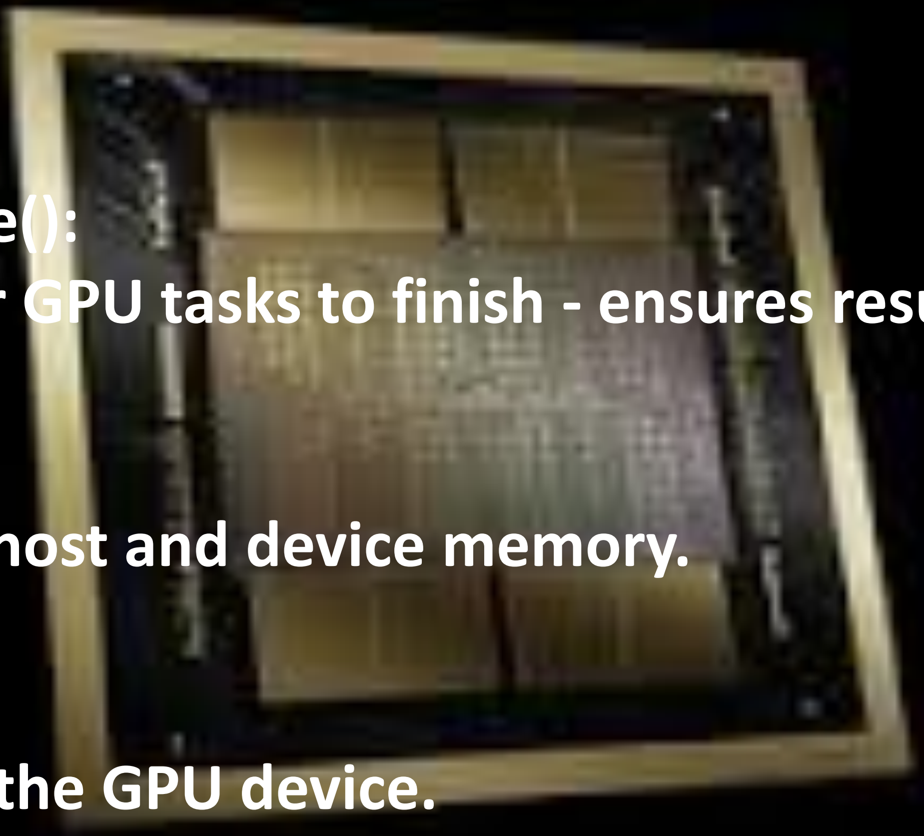
threadIdx.x:
-Built-in variable: thread’s index within its block (x-dimension).

blockIdx.x:
-Built-in variable: Block’s index within the grid (x-dimension).

blockDim.x:
-Built-in variable: number of threads per block (x-dimension).



CUDA Memory and Synchronization



cudaDeviceSynchronize():
-Host waits for all prior GPU tasks to finish - ensures result completion.

cudaMemcpy():
-Copies data between host and device memory.

cudaMalloc():
-Allocates memory on the GPU device.

CUDA Memory and Synchronization

cudaFree():
-Frees memory previously allocated on the device.

cudaGetLastError():
-Returns the last CUDA runtime error that occurred.

cudaGetErrorString():
-Converts an error code into a human-readable error message.

CUDA Memory and Synchronization

cudaDeviceReset():
-Resets the GPU device and flushes outputs - often done at program end.

cudaGetDeviceProperties():
-allows querying device properties.

cudaGetDeviceCount():
-gives number of GPUs available.

KERNEL(device function)

Main (host function)

```
#include<stdio.h>
#include<cuda_runtime.h>

__global__ void helloWorldKernel(){
        printf("Hello World from GPU!\n");
        }

int main(){
        helloWorldKernel<<<1,5>>>();

        cudaDeviceSynchronize();
        return 0;
}
~
```

OUTPUT:-

```
(qdenoise_gpu) shreya@user:~$ nvcc hello.cu -o hello
nvcc warning : Support for offline compilation for architectures prior to '<comp
ute/sm/lto>_75' will be removed in a future release (Use -Wno-deprecated-gpu-tar
gets to suppress warning).
(qdenoise_gpu) shreya@user:~$ ./hello
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
(qdenoise_gpu) shreya@user:~$
```

THE CODE

```
#include<stdio.h>
#include<cuda_runtime.h>

__global__ void helloWorldKernel(){
        printf("Hello World from thread %d and block %d\n", threadIdx.x, blockIdx.x);
        }

int main(){
        helloWorldKernel<<<1,5>>>();

        cudaDeviceSynchronize();
        return 0;
}
```

```
(qdenoise_gpu) shreya@user:~$ ./hello
Hello World from thread 0 and block 0
Hello World from thread 1 and block 0
Hello World from thread 2 and block 0
Hello World from thread 3 and block 0
Hello World from thread 4 and block 0
(qdenoise_gpu) shreya@user:~$ vim hello.cu
(qdenoise_gpu) shreya@user:~$
```

THE OUTPUT

THE CODE

```
#include<stdio.h>
#include<cuda_runtime.h>

__global__ void helloWorldKernel(){
        printf("Hello World from thread %d and block %d\n", threadIdx.x, blockIdx.x);
        }

int main(){
        helloWorldKernel<<<2,5>>>();

        cudaDeviceSynchronize();
        return 0;
}
```

```
(qdenoise_gpu) shreya@user:~$ nvcc hello.cu -o hello
nvcc warning : Support for offline compilation for architectures prior to '<compute/sm/lto>_75'
 will be removed in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).
(qdenoise_gpu) shreya@user:~$ ./hello
Hello World from thread 0 and block 0
Hello World from thread 1 and block 0
Hello World from thread 2 and block 0
Hello World from thread 3 and block 0
Hello World from thread 4 and block 0
Hello World from thread 0 and block 1
Hello World from thread 1 and block 1
Hello World from thread 2 and block 1
Hello World from thread 3 and block 1
```

OUTPUT

```c
#include <stdio.h>
#include<cuda_runtime.h>

// CUDA kernel for vector addition
__global__ void vectorAdd(const float *A, const float *B, float *C, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    int N = 1000;
    size_t size = N * sizeof(float);
    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C = (float*)malloc(size);
    for (int i = 0; i < N; i++) {
        h_A[i] = 1.0f;
        h_B[i] = 2.0f;
    }

    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    for (int i = 0; i < 5; i++) {
        printf("%f + %f = %f\n", h_A[i], h_B[i], h_C[i]);
    }
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);
    return 0;
}
-- INSERT --
```

```
(qdenoise_gpu) shreya@user:~$ vim vecAdd.cu
(qdenoise_gpu) shreya@user:~$ nvcc vecAdd.cu -o vecAdd


nvcc warning : Support for offline compilation for architectures prior to '<comp
ute/sm/lto>_75' will be removed in a future release (Use -Wno-deprecated-gpu-tar
gets to suppress warning).
(qdenoise_gpu) shreya@user:~$ ./vecAdd
1.000000 + 2.000000 = 3.000000
1.000000 + 2.000000 = 3.000000
1.000000 + 2.000000 = 3.000000
1.000000 + 2.000000 = 3.000000
1.000000 + 2.000000 = 3.000000
(qdenoise_gpu) shreya@user:~$
```

CUDA Variable qualifiers:

__shared__ :

- Purpose: Declares a variable that resides in shared memory within a thread block. Shared memory is on-chip, offering very low latency access for threads within the same block.
- Accessibility: Accessible by all threads within the same thread block.
- Lifetime: Has the lifetime of the thread block.
- Example: __shared__ float sharedArray[256]; (static allocation) or extern __shared__ float dynamicSharedArray(); (dynamic allocation)

CUDA Variable qualifiers:

__constant__ :

- Purpose: Declares a variable that resides in constant memory on the device. Constant memory is read-only for device kernels and is typically cached, leading to faster access for all threads when data is uniform.
- Accessibility: Accessible by all threads across the entire grid and by the host through the CUDA runtime library.
- Lifetime: Has the lifetime of the application.
- Example: __constant__ float constantValue = 3.14f;

CUDA Variable qualifiers:

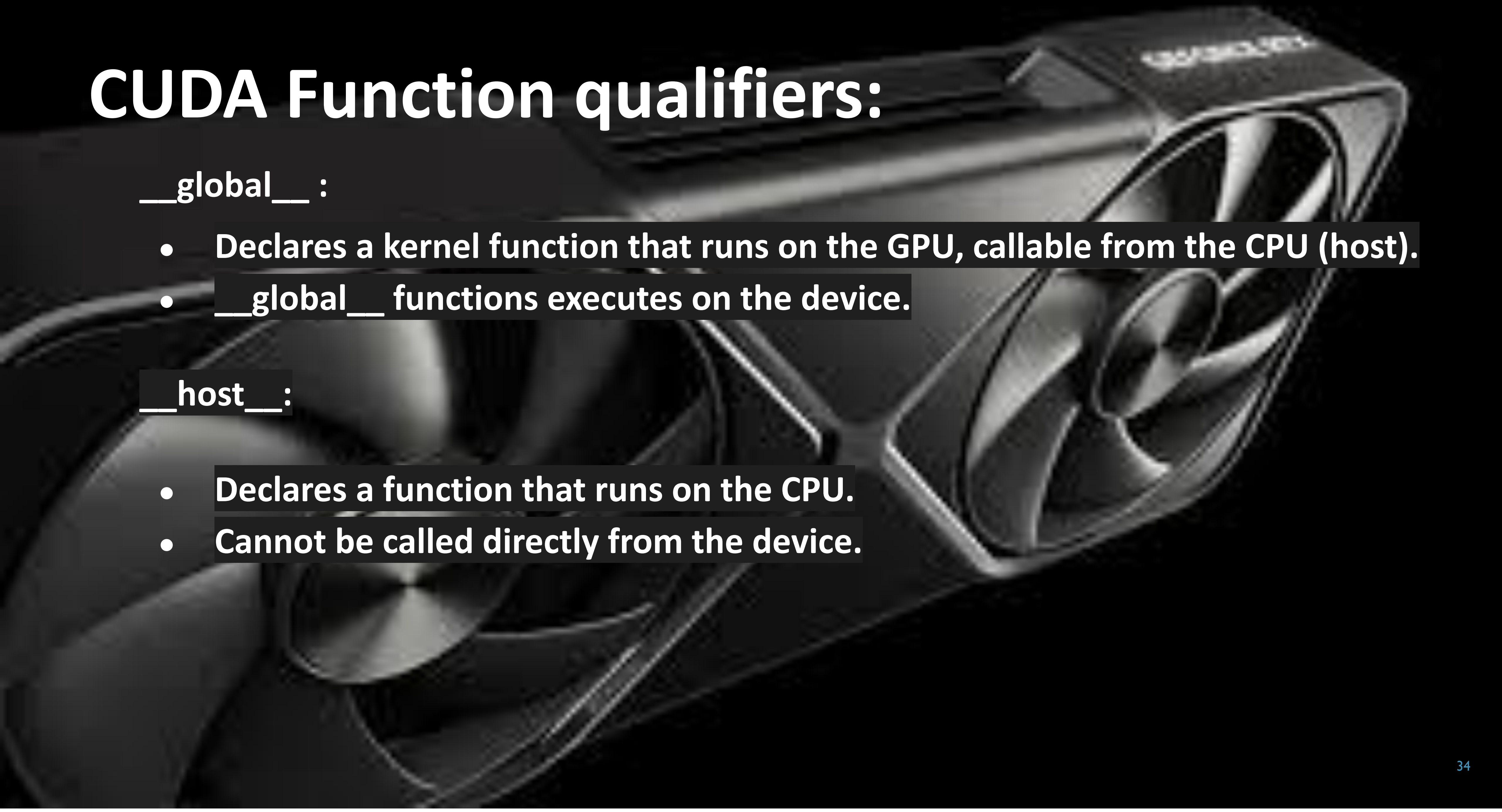
__device__ :

- Purpose: Declares a variable that resides in the global memory of the device.
- Accessibility: Accessible by all threads across the entire grid and by the host through the CUDA runtime library.
- Lifetime: Has the lifetime of the application.
- Example: __device__ float globalVariable;

CUDA Variable qualifiers:

__managed__ :

- Purpose: Declares a variable that resides in the global memory of the device.
- Accessibility: Accessible by all threads across the entire grid and by the host through the CUDA runtime library.
- Lifetime: Has the lifetime of the application.
- Example: __device__ float globalVariable;



CUDA Function qualifiers:

__global__ :

- Declares a kernel function that runs on the GPU, callable from the CPU (host).
- __global__ functions executes on the device.

__host__:

- Declares a function that runs on the CPU.
- Cannot be called directly from the device.

CUDA Function qualifiers:

__device__ :

- Declares a function that runs or resides on the GPU, callable from device code.
- Cannot be called directly from host.

__host__ __device__:

- Declares a function callable from both CPU and GPU, allowing code reuse.

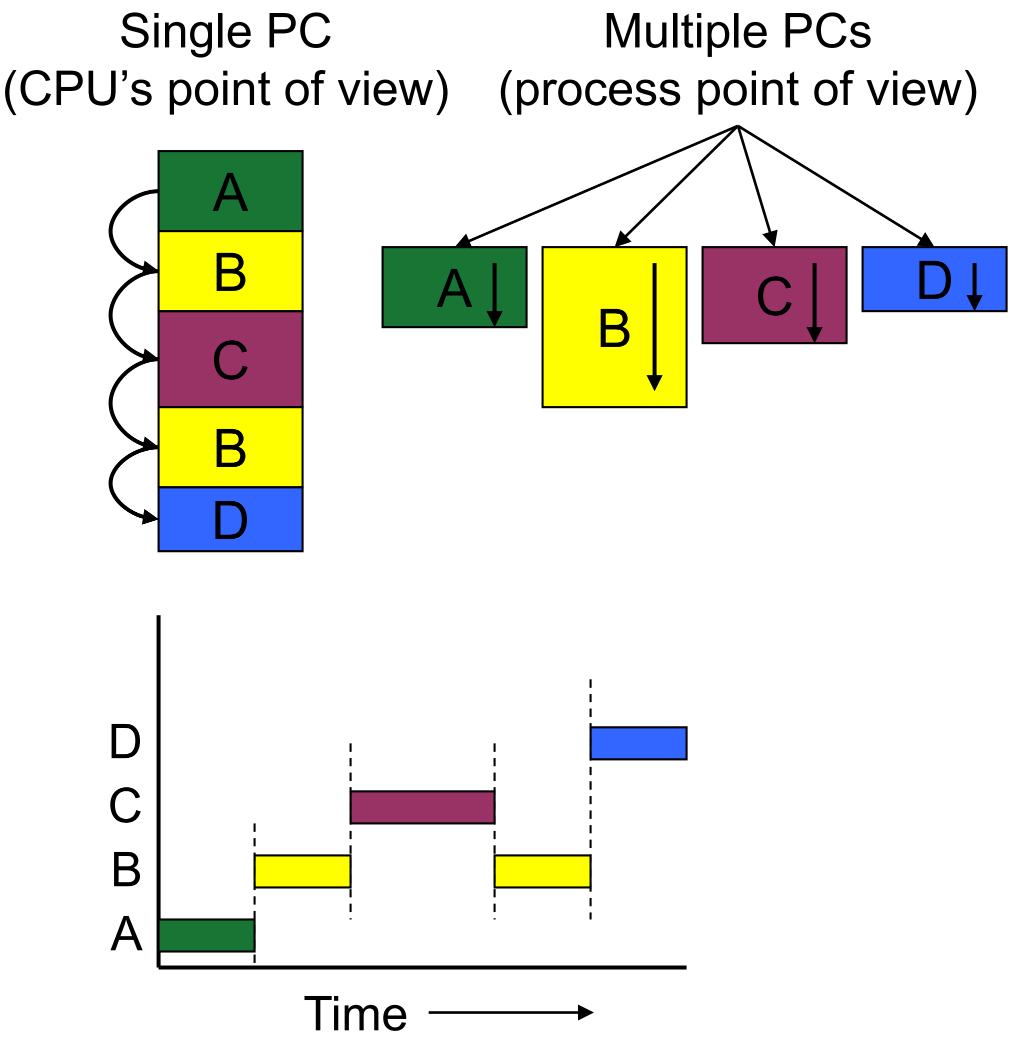
CPU : PROCESSES & THREADS

PROCESSES AND THREADS

- Processes
- Threads
- Scheduling
- Interprocess communication
- Classical IPC problems

WHAT IS A PROCESS?

- Code, data, and stack
  - Usually (but not always) has its own address space
- Program state
  - CPU registers
  - Program counter (current location in the code)
  - Stack pointer
- Only one process can be running in the CPU at any given time!



- Multiprogramming of four programs
- Conceptual model
  - 4 independent processes
  - Processes run sequentially
- Only one program active at any instant!
  - That instant can be very short…

WHEN IS A PROCESS CREATED?

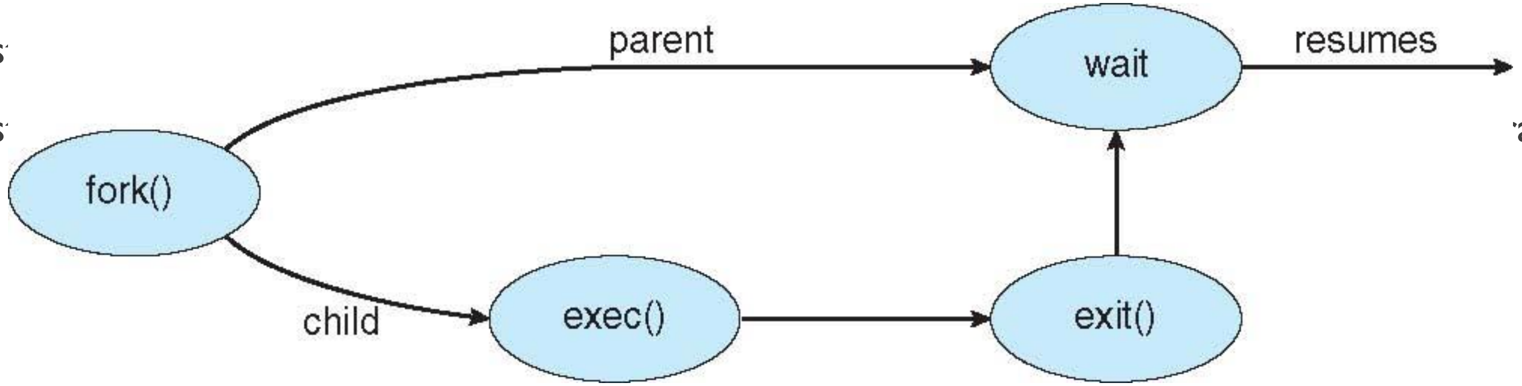
- Processes can be created in two ways
  - System initialization: one or more processes created when the OS starts up
  - Execution of a process creation system call: something explicitly asks for a new process
- System calls can come from
  - User request to create a new process (system call executed from user shell)
  - Already running processes
    - User programs
    - System daemons

PROCESS CREATION

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

PROCESS CREATION (CONT.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** sys
  - **`exec()`** sys ... am



WHEN DO PROCESSES END?

- Conditions that terminate processes can be
  - Voluntary
  - Involuntary
- Voluntary
  - Normal exit
  - Error exit
- Involuntary
  - Fatal error (only sort of involuntary)
  - Killed by another process

PROCESS TERMINATION

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

PROCESS TERMINATION

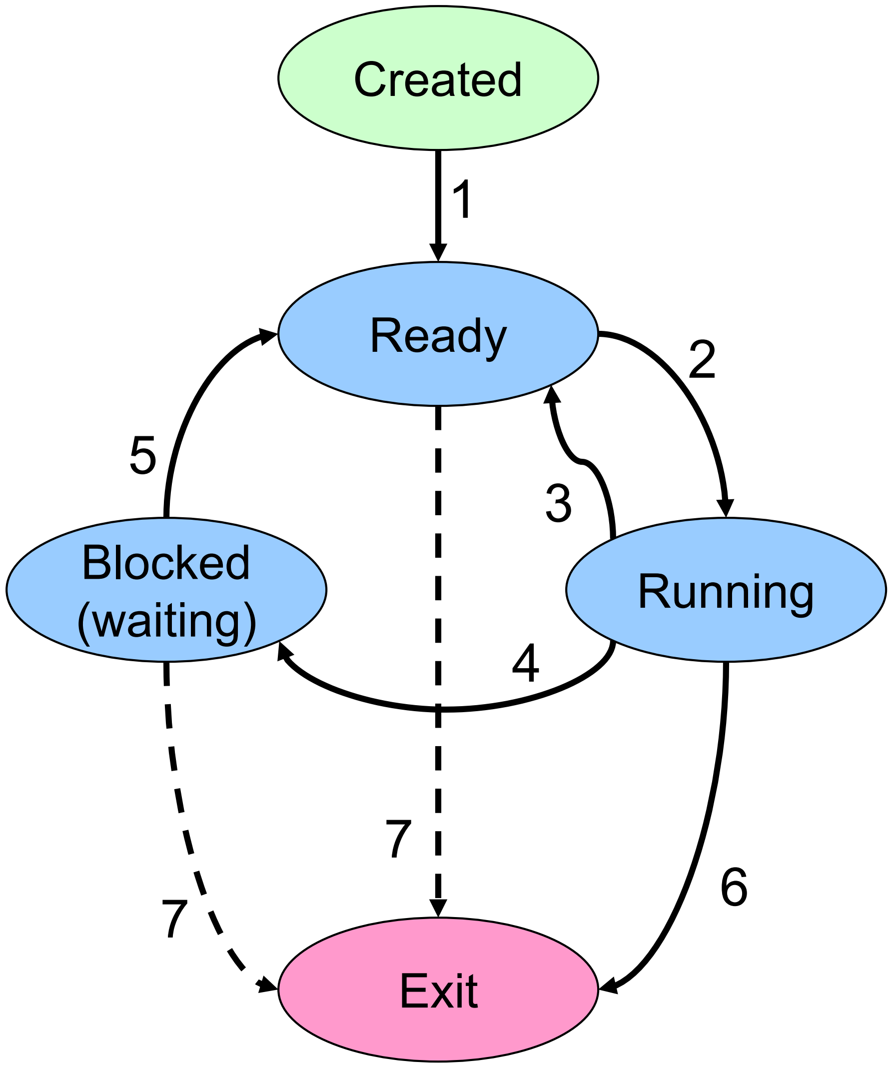
- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait`**, process is an **orphan**

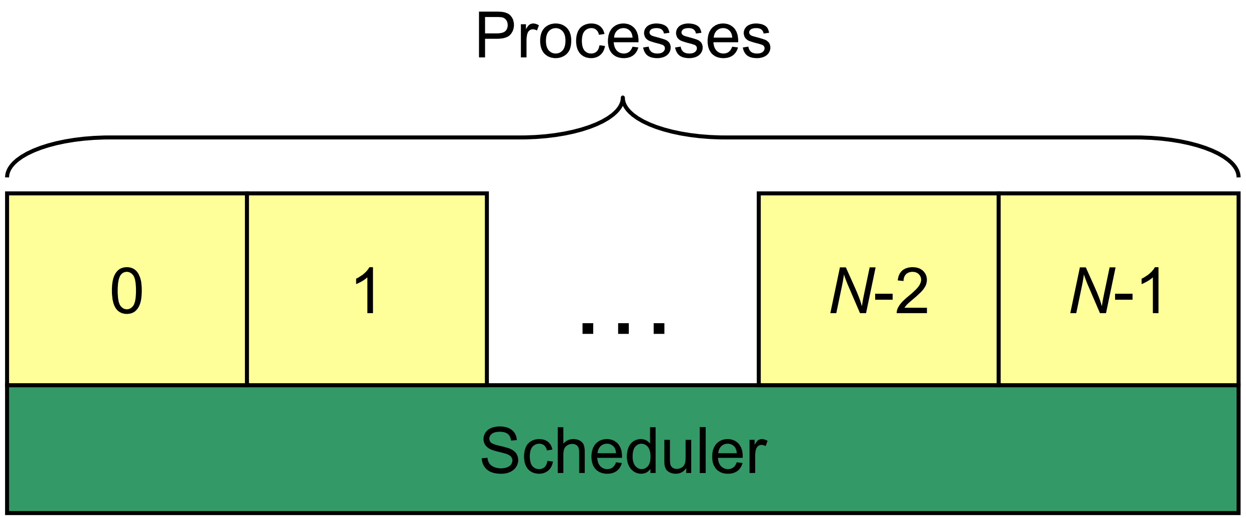
PROCESS HIERARCHIES

- Parent creates a child process
  - Child processes can create their own children
- Forms a hierarchy
  - UNIX calls this a "process group"
  - If a process exits, its children are "inherited" by the exiting process's parent
- Windows has no concept of process hierarchy
  - All processes are created equal



- Process in one of 5 states
  - Created
  - Ready
  - Running
  - Blocked
  - Exit
- Transitions between states
  1 - Process enters ready queue
  2 - Scheduler picks this process
  3 - Scheduler picks a different process
  4 - Process waits for event (such as I/O)
  5 - Event occurs
  6 - Process exits
  7 - Process ended by another process

- Two “layers” for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
  - Processes tracked in the *process table*
  - Each process has a *process table entry*



WHAT'S IN A PROCESS TABLE ENTRY?

May be stored on stack: Registers, Program counter, CPU status word

Process management

- Registers
- Program counter
- CPU status word
- Stack pointer
- Process state
- Priority / scheduling parameters
- Process ID
- Parent process ID
- Signals
- Process start time
- Total CPU usage

File management

- Root directory
- Working (current) directory
- File descriptors
- User ID
- Group ID

Memory management

Pointers to text, data, stack

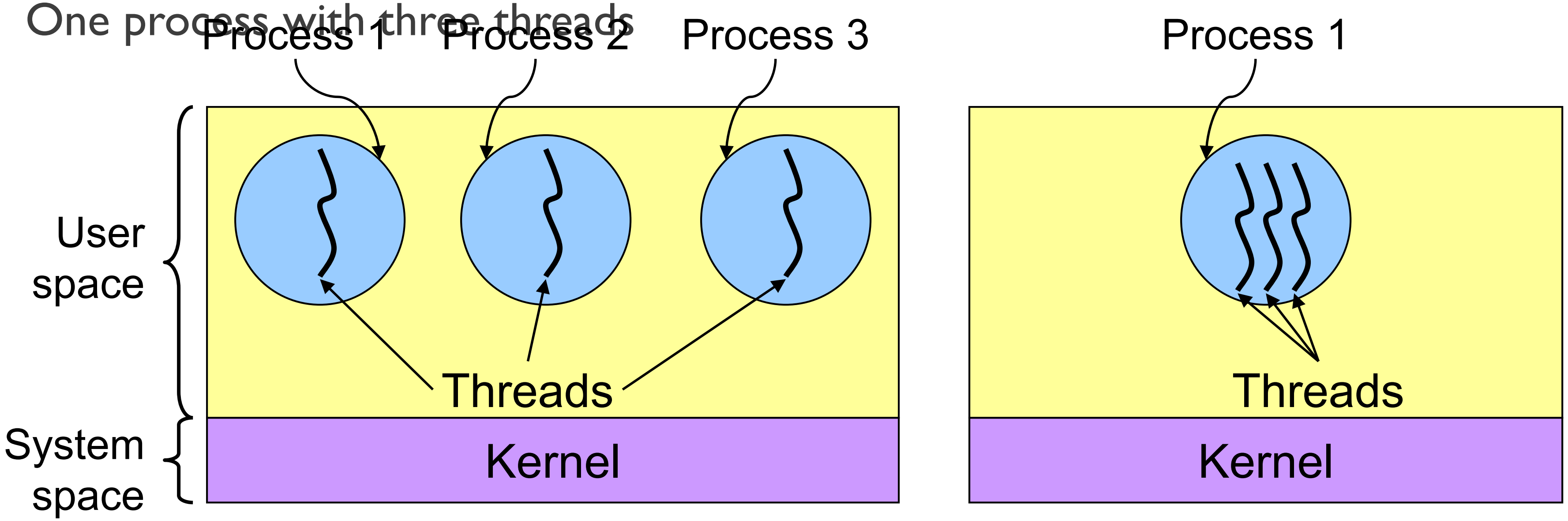
or

Pointer to page table

WHAT HAPPENS ON A TRAP/INTERRUPT?

1. Hardware saves program counter (on stack or in a special register)
2. Hardware loads new PC, identifies interrupt
3. Assembly language routine saves registers
4. Assembly language routine sets up stack
5. Assembly language calls C to run service routine
6. Service routine calls scheduler
7. Scheduler selects a process to run next (might be the one interrupted…)
8. Assembly language routine loads PC & registers for the selected process

- Process == address space
- Thread == program counter / stream of instructions
- Two examples
  - Three processes, each with one thread
  - One process with three threads

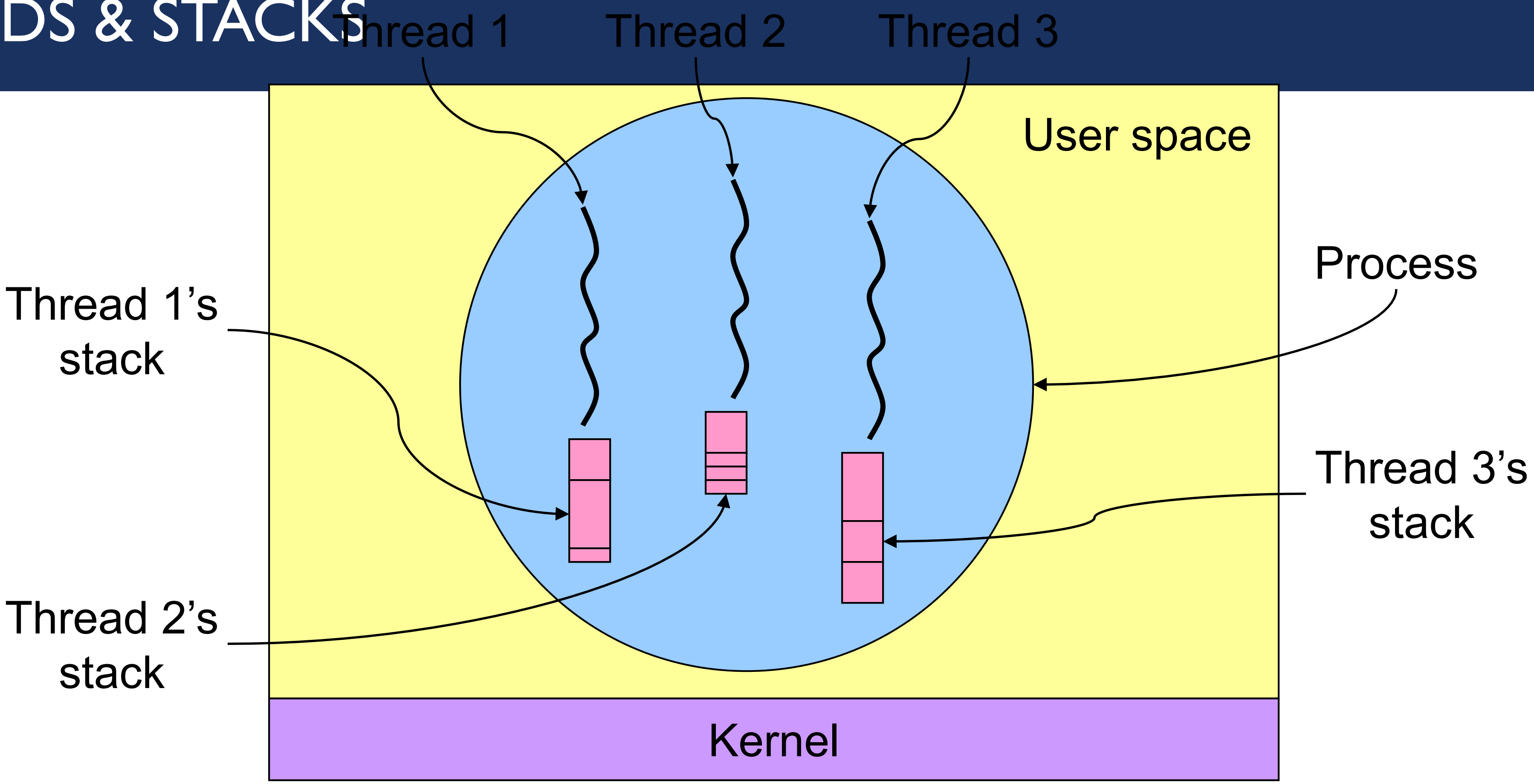


PROCESS & THREAD INFORMATION

Per process items
Address space
Open files
Child processes
Signals & handlers
Accounting info
Global variables

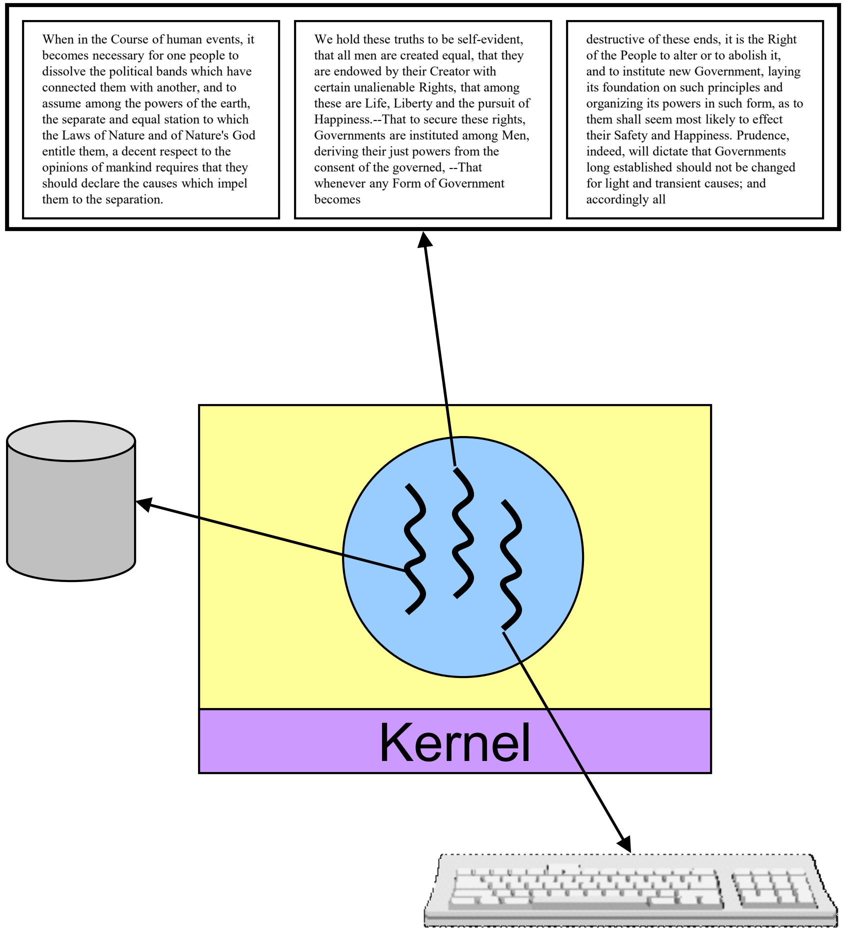
Per thread items	**Per thread items**	**Per thread items**
Program counter	Program counter	Program counter
Registers	Registers	Registers
Stack & stack pointer	Stack & stack pointer	Stack & stack pointer
State	State	State

THREADS & STACKS

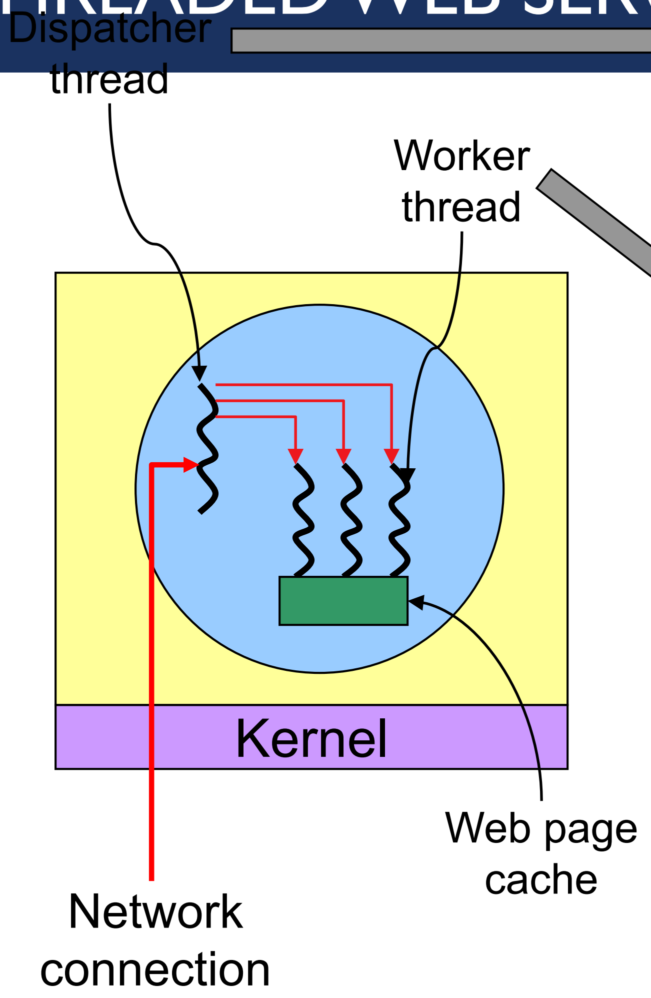


=> Each thread has its own stack!

- Allow a single application to do many things at once
  - Simpler programming model
  - Less waiting
- Threads are faster to create or destroy
  - No separate address space
- Overlap computation and I/O
  - Could be done without threads, but it's harder
- Example: word processor
  - Thread to read from keyboard
  - Thread to format document
  - Thread to write to disk



MULTITHREADED WEB SERVER



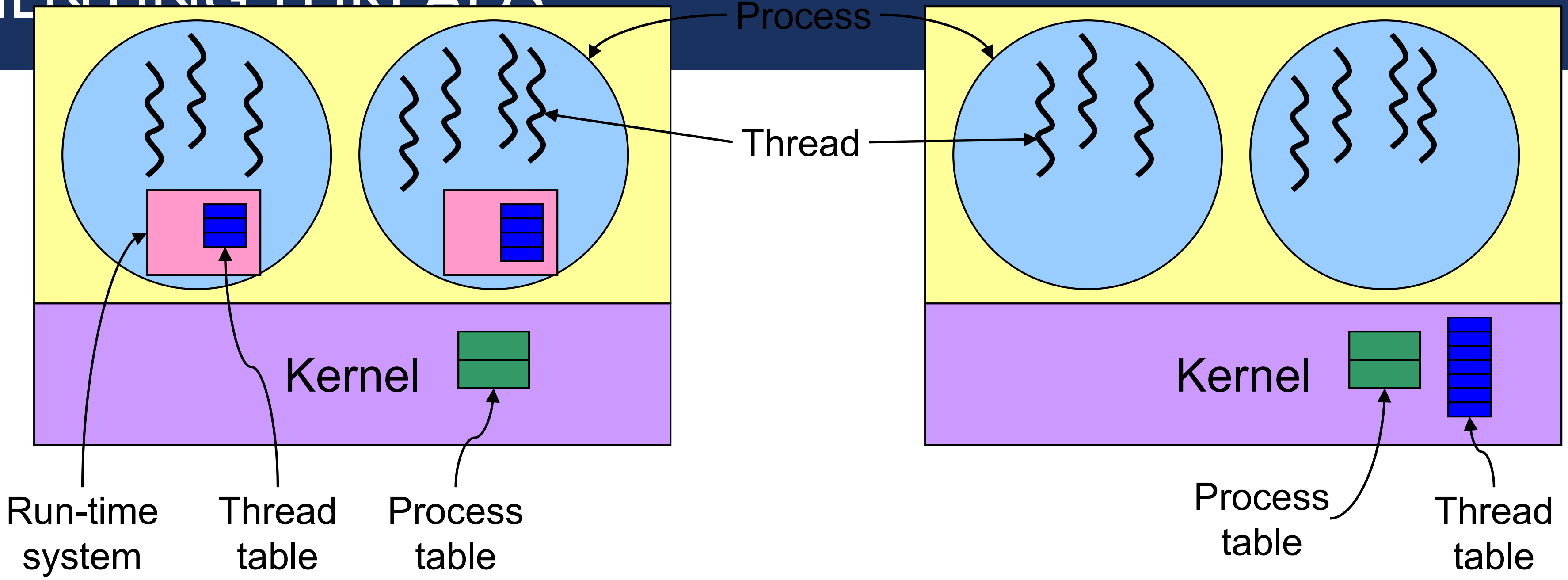

```
while(TRUE) {
  getNextRequest(&buf);
  handoffWork(&buf);

}
```

```
while(TRUE) {
  waitForWork(&buf);
  lookForPageInCache(&buf,&page);
  if(pageNotInCache(&page)) {
    readPageFromDisk(&buf,&page);
  }
  returnPage(&page);

}
```

THREE WAYS TO BUILD A SERVER

- Thread model
  - Parallelism
  - Blocking system calls
- Single-threaded process: slow, but easier to do
  - No parallelism
  - Blocking system calls
- Finite-state machine
  - Each activity has its own state
  - States change when system calls complete or interrupts occur
  - Parallelism
  - Nonblocking system calls
  - Interrupts

IMPLEMENTING THREADS



User-level threads
+ No need for kernel support
- May be slower than kernel threads
- Harder to do non-blocking I/O

Kernel-level threads
+ More flexible scheduling
+ Non-blocking I/O
- Not portable

INTERPROCESS COMMUNICATION

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

PRODUCER-CONSUMER PROBLEM

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

BOUNDED-BUFFER – SHARED-MEMORY SOLUTION

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

BOUNDED-BUFFER – PRODUCER

```
item next_produced;
while (true) {
      /* produce an item in next produced */
      while (((in + 1) % BUFFER_SIZE) == out)
            ; /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```

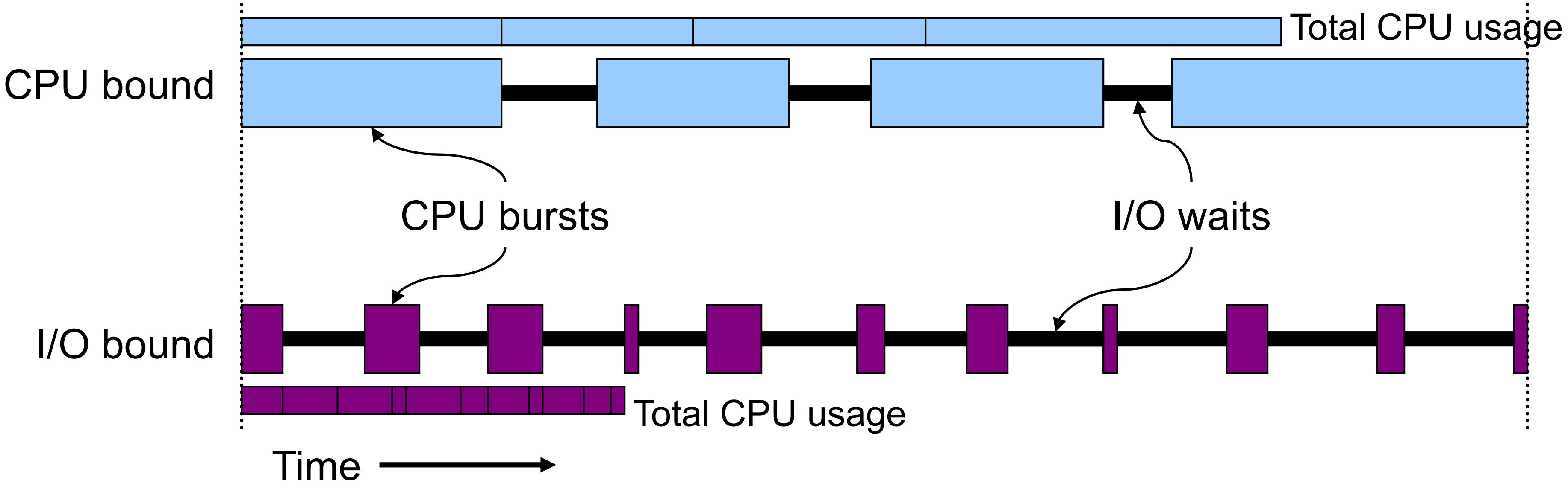
BOUNDED BUFFER – CONSUMER

```
item next_consumed;

while (true) {
   while (in == out)

      ; /* do nothing */
   next_consumed = buffer[out];

   out = (out + 1) % BUFFER_SIZE;


   /* consume the item in next consumed */

}
```

SCHEDULING

- What is scheduling?
  - Goals
  - Mechanisms
- Scheduling on batch systems
- Scheduling on interactive systems
- Other kinds of scheduling
  - Real-time scheduling

- Bursts of CPU usage alternate with periods of I/O wait
- Some processes are *CPU-bound*: they don't make many I/O requests
- Other processes are *I/O-bound* and make many kernel requests



WHEN ARE PROCESSES SCHEDULED?

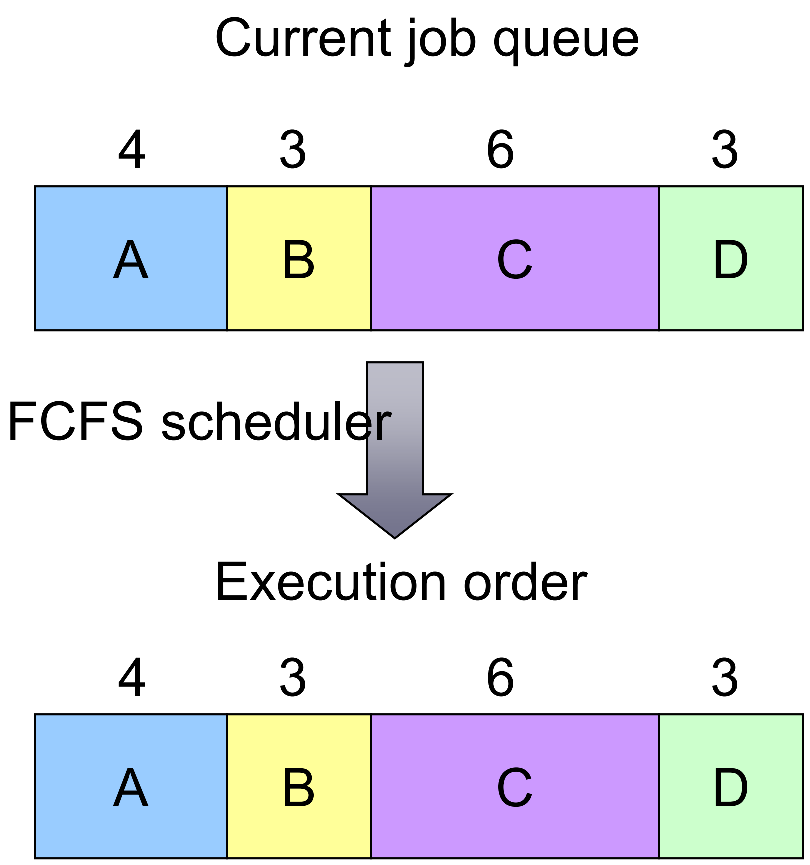
- At the time they enter the system
  - Common in batch systems
  - Two types of batch scheduling
    - Submission of a new job causes the scheduler to run
    - Scheduling only done when a job voluntarily gives up the CPU (*i.e.*, while waiting for an I/O request)
- At relatively fixed intervals (clock interrupts)
  - Necessary for interactive systems
  - May also be used for batch systems
  - Scheduling algorithms at each interrupt, and picks the next process from the pool of "ready" processes

SCHEDULING GOALS

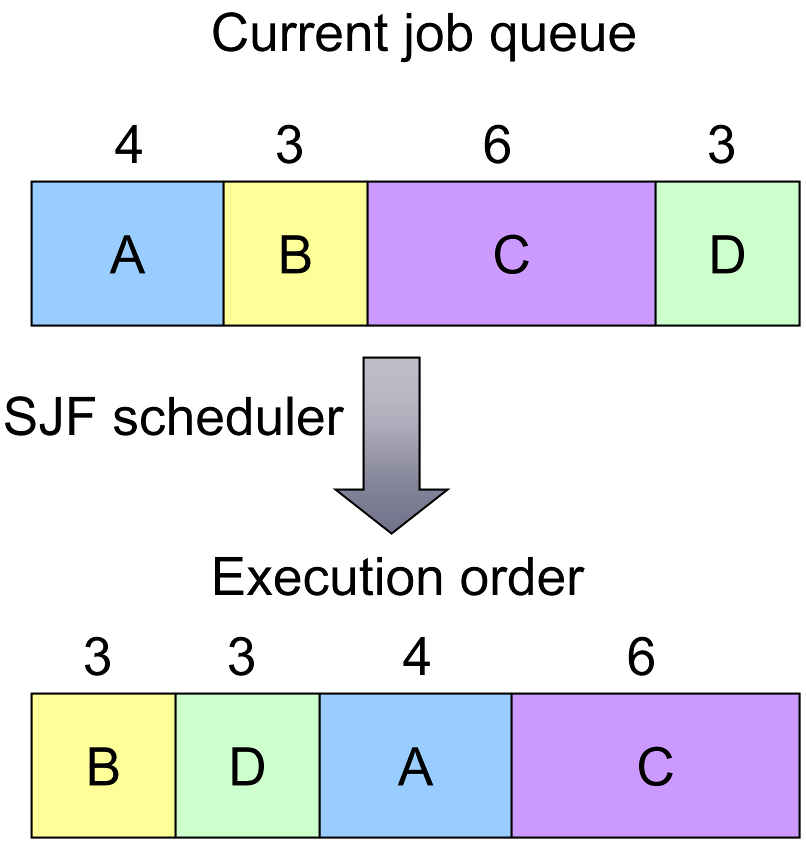
- All systems
  - Fairness: give each process a fair share of the CPU
  - Enforcement: ensure that the stated policy is carried out
  - Balance: keep all parts of the system busy
- Batch systems
  - Throughput: maximize jobs per unit time (hour)
  - Turnaround time: minimize time users wait for jobs
  - CPU utilization: keep the CPU as busy as possible
- Interactive systems
  - Response time: respond quickly to users' requests
  - Proportionality: meet users' expectations
- Real-time systems
  - Meet deadlines: missing deadlines is a system failure!
  - Predictability: same type of behavior for each time slice

MEASURING SCHEDULING PERFORMANCE

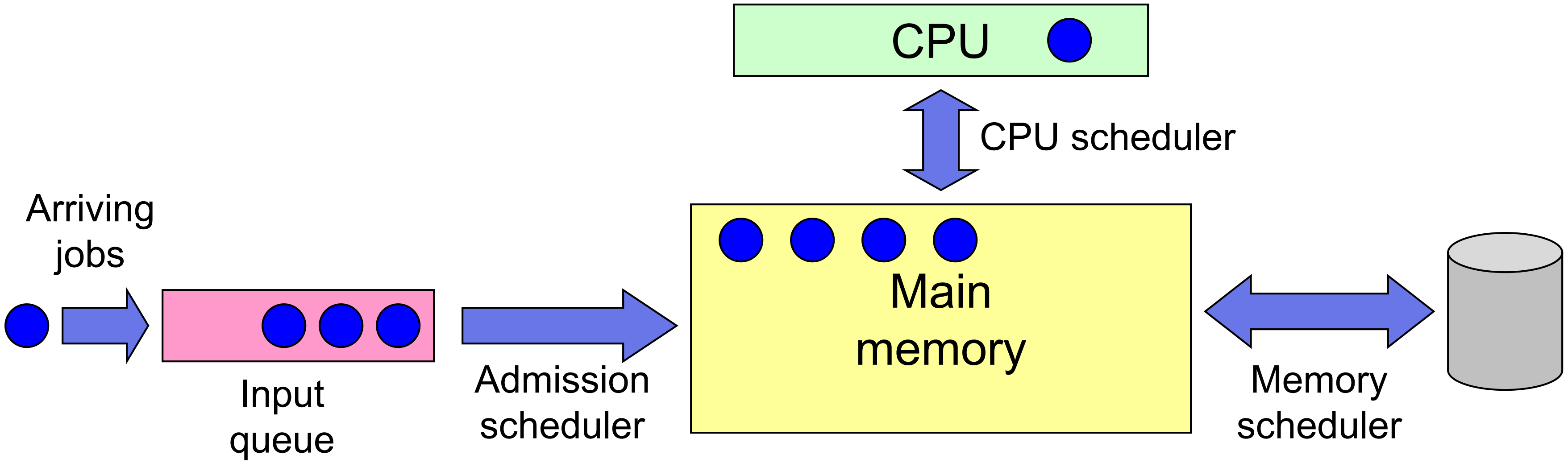
- Throughput
  - Amount of work completed per second (minute, hour)
  - Higher throughput usually means better utilized system
- Response time
  - Response time is time from when a command is submitted until results are returned
  - Can measure average, variance, minimum, maximum, …
  - May be more useful to measure time spent waiting
- Turnaround time
  - Like response time, but for batch jobs (response is the completion of the process)
- Usually not possible to optimize for *all* metrics with the same scheduling algorithm



- Goal: do jobs in the order they arrive
  - Fair in the same way a bank teller line is fair
- Simple algorithm!
- Problem: long jobs delay every job after them
  - Many processes may wait for a single long job

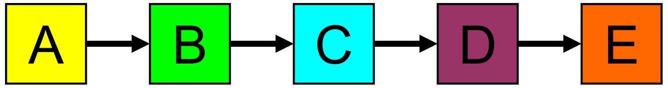


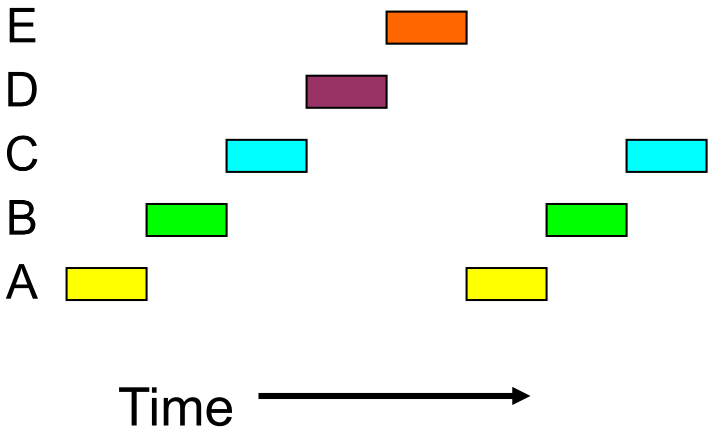
- Goal: do the shortest job first
  - Short jobs complete first
  - Long jobs delay every job after them
- Jobs sorted in increasing order of execution time
  - Ordering of ties doesn't matter
- Shortest Remaining Time First (SRTF): preemptive form of SJF
- Problem: how does the scheduler know how long a job will take?



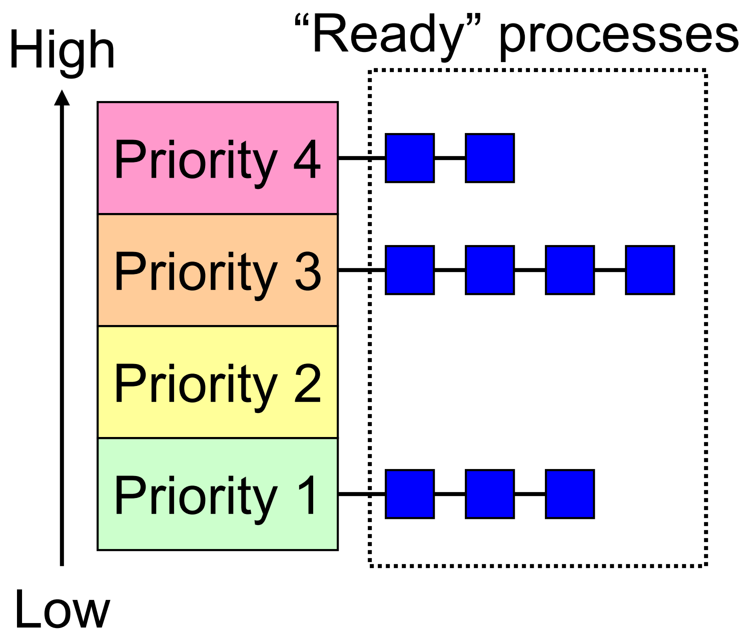
- Jobs held in input queue until moved into memory
  - Pick "complementary jobs": small & large, CPU- & I/O-intensive
  - Jobs move into memory when admitted
- CPU scheduler picks next job to run
- Memory scheduler picks some jobs from main memory and moves them to disk if insufficient memory space

- Round Robin scheduling
  - Give each process a fixed time slot (*quantum*)
  - Rotate through "ready" processes
  - Each process makes some progress
- What's a good quantum?
  - Too short: many process switches hurt efficiency
  - Too long: poor response to interactive requests
  - Typical length: 10–50 ms





- Assign a priority to each process
  - "Ready" process with highest priority allowed to run
  - Running process may be interrupted after its quantum expires
- Priorities may be assigned dynamically
  - Reduced when a process uses CPU time
  - Increased when a process waits for I/O
- Often, processes grouped into multiple queues based on priority, and run round-robin per queue



SHORTEST PROCESS NEXT

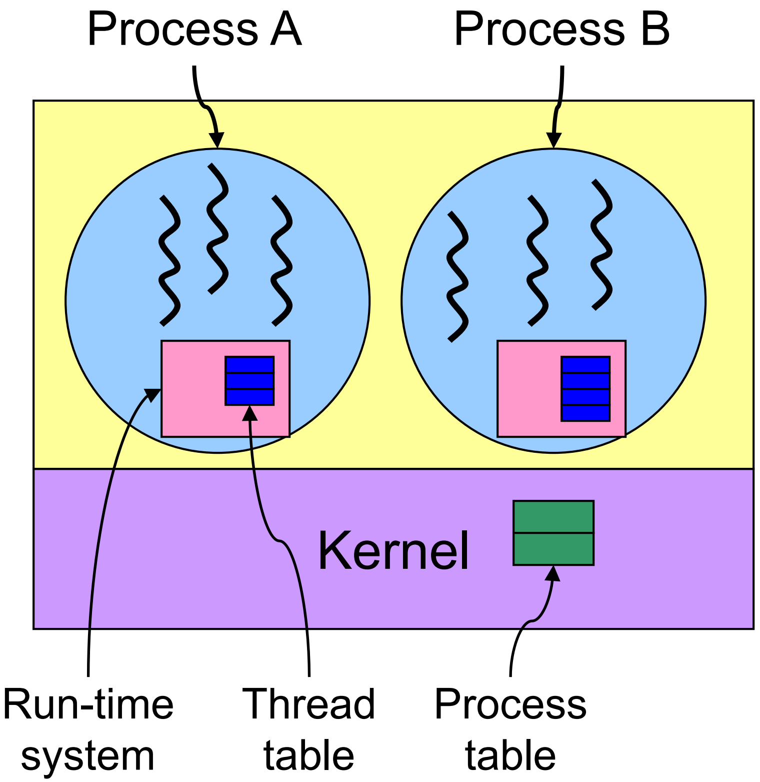
- Run the process that will finish the soonest
  - In interactive systems, job completion time is unknown!
- Guess at completion time based on previous runs
  - Update estimate each time the job is run
  - Estimate is a combination of previous estimate and most recent run time
- Not often used because round robin with priority works so well!

LOTTERY SCHEDULING

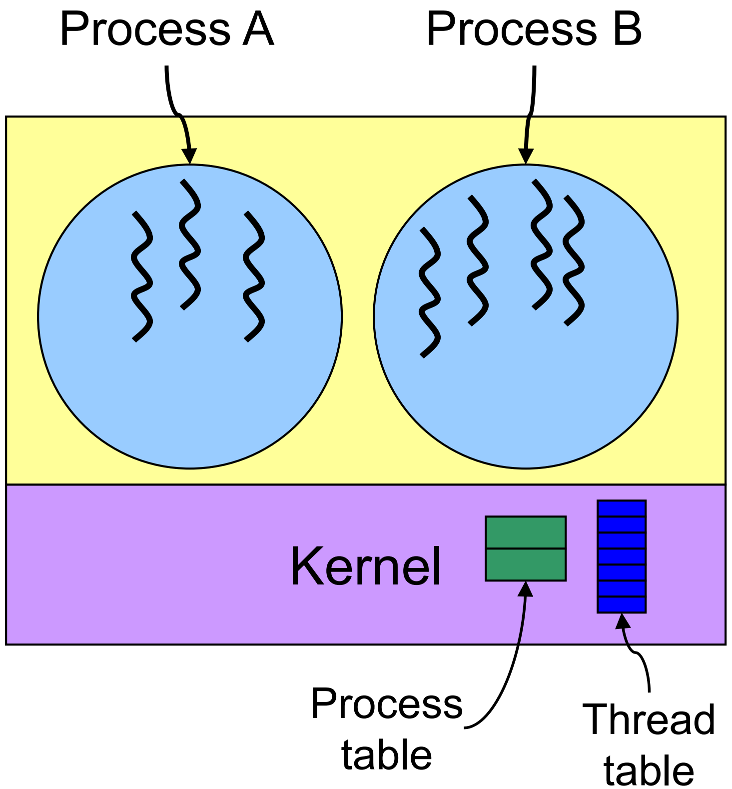
- Give processes "tickets" for CPU time
  - More tickets => higher share of CPU
- Each quantum, pick a ticket at random
  - If there are *n* tickets, pick a number from 1 to *n*
  - Process holding the ticket gets to run for a quantum
- Over the long run, each process gets the CPU *m*/*n* of the time if the process has *m* of the *n* existing tickets
- Tickets can be transferred
  - Cooperating processes can exchange tickets
  - Clients can transfer tickets to server so it can have a higher priority

POLICY VERSUS MECHANISM

- Separate what ***may*** be done from ***how*** it is done
  - Mechanism allows
    - Priorities to be assigned to processes
    - CPU to select processes with high priorities
  - Policy set by what priorities are assigned to processes
- Scheduling algorithm parameterized
  - Mechanism in the kernel
  - Priorities assigned in the kernel or by users
- Parameters may be set by user processes
  - Don't allow a user process to take over the system!
  - Allow a user process to voluntarily lower its own priority
  - Allow a user process to assign priority to its threads



- Kernel picks a process to run next
- Run-time system (at user level) schedules threads
  - Run each thread for less than process quantum
  - Example: processes get 40ms each, threads get 10ms each
- Example schedule: A1,A2,A3,A1,B1,B3,B2,B3
- Not possible: A1,A2,B1,B2,A3,B3,A2,B1



- Kernel schedules each thread
  - No restrictions on ordering
  - May be more difficult for each process to specify priorities
- Example schedule: A1,A2,A3,A1,B1,B3,B2,B3
- Also possible: A1,A2,B1,B2,A3,B3,A2,B1

ASSIGNMENTS

Phase	Focus	Analysis Tool	Compare With
CPU Phase 1A	Serial C++	perf, gprof	Baseline
CPU Phase 1B	Multi-Threaded C++	perf, htop, OpenMP stats	Serial
GPU Phase 2	Same Assignments but executed in GPU	gprop, FSight	CPU, TPU,DPU, QBit

Tool	Purpose	Example Command
time ./program	Total wall time	Basic timing
perf stat ./program	CPU cycles, cache misses, thread context switches	System-level metrics
htop / top -H	Live per-thread CPU usage	Observe thread load
valgrind --tool=callgrind ./program	Function call and instruction profile	Analyze bottlenecks
omp_get_wtime()	Measure elapsed time in OpenMP	Fine-grained measurement

ASSIGNMENTS REPORT/ ANALYSIS EXAMPLE

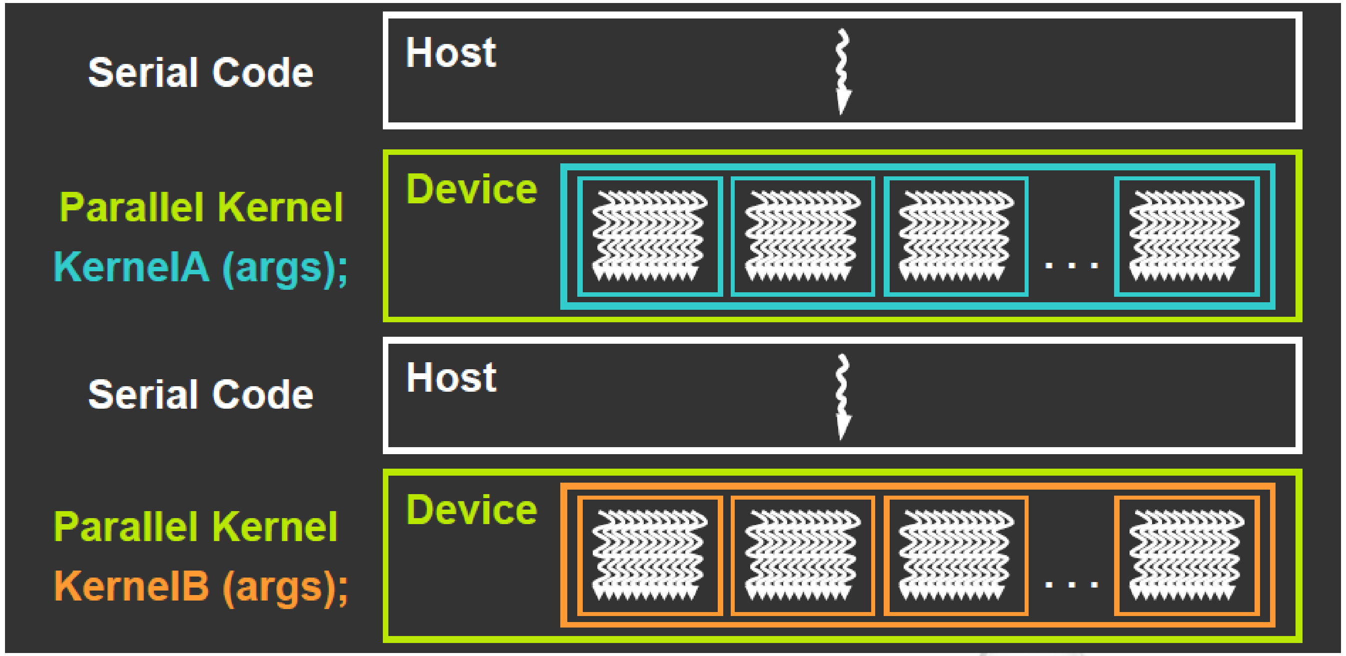
Same columns to be created and filled with GPU/TPU/DPU/Qbit

Assignment #	Program Description	Tool Used (`time`, `perf`, `valgrind`, etc.)	Execution Time (sec)	CPU Utilization (%)	Cache Misses	Threads/Processes Used	Observations / Comments
1	Vector Addition (CPU)	`perf stat ./a.out`					
2	Matrix Multiplication (CPU)	`time ./matrix_mult`					
3	Image Blur Filter	`valgrind --tool=cachegrind ./a.out`					
4	Multi-threaded Vector Add	`perf stat ./a.out`					
5	OpenMP Matrix Multiplication	`perf stat ./omp_matrix_mult`					

GPU programming: CUDA programming

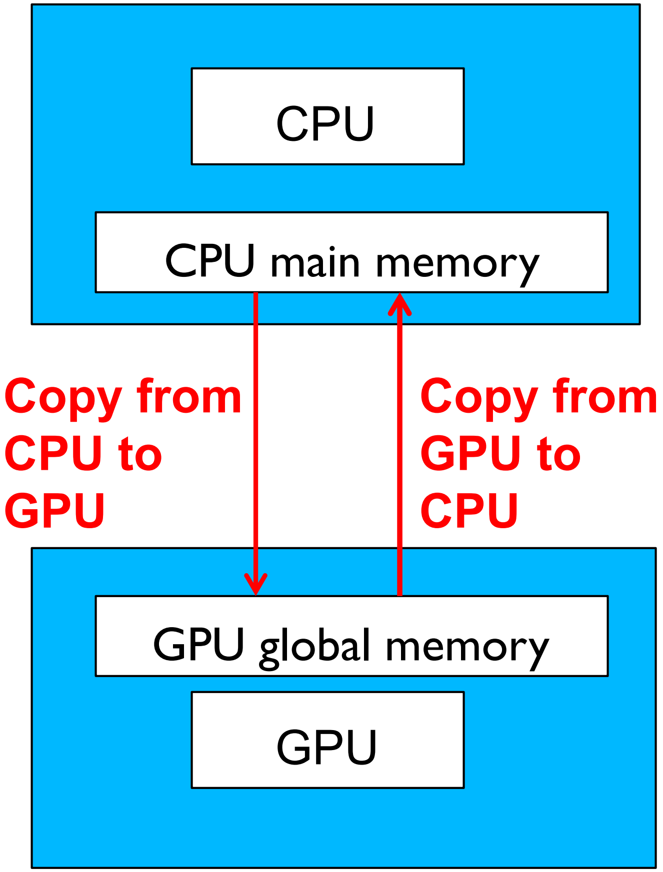
- CUDA is Nvidia's scalable parallel programming model and a software environment for parallel computing that allows the use of GPU for general purpose processing.
  - Language: CUDA C, minor extension to C/C++
    - Let the programmer focus on parallel algorithms not parallel programming mechanisms.
  - A heterogeneous serial-parallel programming model
    - Designed to program heterogeneous CPU+GPU systems
      - CPU and GPU are separate devices with separate memory

- Fork-join model: CUDA program = serial code + parallel kernels (all in CUDA C)
  - Serial C code executes in a host thread (CPU thread)
  - Parallel kernel code executes in many device threads (GPU threads)



- Kernel code is regular C code except that it will use thread ID (CUDA built-in variable) to make different threads operate on different data
    - Also have variables for the total number of threads
- When a kernel is reached in the code for the first time, it is launched onto GPU.

- CPU and GPU have different memories:
  - CPU memory is called host memory
  - GPU memory is called device memory
  - Implication:
    - Explicitly transfer data from CPU to GPU for GPU computation, and
    - Explicitly transfer results in GPU memory copied back to CPU memory



```
int main (int argc, char **argv ) {
        1. Allocate memory space in device (GPU) for data
        2. Allocate memory space in host (CPU) for data

        3. Copy data to GPU

        4. Call “kernel” routine to execute on GPU
        (with CUDA syntax that defines no of threads and their physical structure)

        5. Transfer results from GPU to CPU

        6. Free memory space in device (GPU)
        7. Free memory space in host (CPU)
        return;
}.
```

- The cudaMalloc routine: allocates object in the device global memory
  - Two parameters:
    - address of a pointer to the allocated object
    - size of the allocated object in terms of bytes.

```
int size = N *sizeof( int);       // space for N integers
int *devA, *devB, *devC;     // devA, devB, devC ptrs
cudaMalloc( (void**)&devA, size) );
cudaMalloc( (void**)&devB, size );
cudaMalloc( (void**)&devC, size );
```

- **2. Allocating memory in host (CPU)?**
  - **The regular malloc routine**

- CUDA routine cudaMemcpy: memory data transfer
  - four parameters
    - pointer to destination
    - pointer to source
    - number of bytes copied
    - Type/direction of transfer

cudaMemcpy(devA, &A, size, cudaMemcpyHostToDevice);

cudaMemcpy(devB, &B, size, cudaMemcpyHostToDevice);

DevA and devB are pointers to destination in device (return from *cudaMalloc* and A and B are pointers to host data

Defining and invoking kernel routine

Define: CUDA specifier __global__

A kernel that can be called from the host.

```
#define N 256

__global__ void vecAdd(int *A, int *B, int *C) {  // Kernel definition
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
}

int main() {
// allocate device memory &
// copy data to device
// device mem. ptrs devA,devB,devC
   vecAdd<<<1, N>>>(devA,devB,devC);
   …
}
```

threadIdx is a built-in variable

Each thread performs one pair-wise addition:

Thread 0: devC[0] = devA[0] + devB[0];
Thread 1: devC[1] = devA[1] + devB[1];
Thread 2: devC[2] = devA[2] + devB[2];

This is the fork-join statement in Cuda
Notice the devA/B/C are device memory pointer

- **<<<…>>>** syntax (addition to C) for kernel calls:

myKernel<<< n, m >>>(arg1, …);

- **<<< … >>>** contains thread organization for this particular kernel call in two parameters, **n** and **m**:
  - **vecAdd<<<1, N>>>(devA,devB,devC): 1 dimension block with N threads in the block.**
    - **Threads execute very efficiently on GPU (much more efficiently than pthread or OpenMP threads): we can have fine-grain threads (a few statements)**
  - **More thread organization later**
- **arg1**, … , -- arguments to routine **myKernel** typically pointers to device memory obtained previously from **cudaMallac**.

- Once the kernel returns, the results are in the GPU memory (dec_C in the example).

- CUDA routine cudaMemcpy

```
cudaMemcpy( &C, dev_C, size, cudaMemcpyDeviceToHost);
```

  - **dev_C** is a pointer in device memory and **C** is a pointer in host memory.

- In “device” (GPU) -- Use CUDA cudaFree routine:

```
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
```

- In (CPU) host (if CPU memory allocated with malloc) -- Use regular C free routine:

```
free( a );
free( b );
free( c );
```

- See lect26/vecadd.cu
- Compiling CUDA programs
  - Use the aurora1.cs.fsu.edu and aurora2.cs.fsu.edu
  - Naming convention: .cu programs are CUDA programs
  - NVIDIA CUDA compiler driver: nvcc
  - To compile vecadd.cu: nvcc –O3 vecadd.cu

- nvcc “wrapper” divides code into host and device parts.
- Host part compiled by regular C compiler
- Device part compiled by NVCC’s runtime component
- Two compiled parts combined into one executable

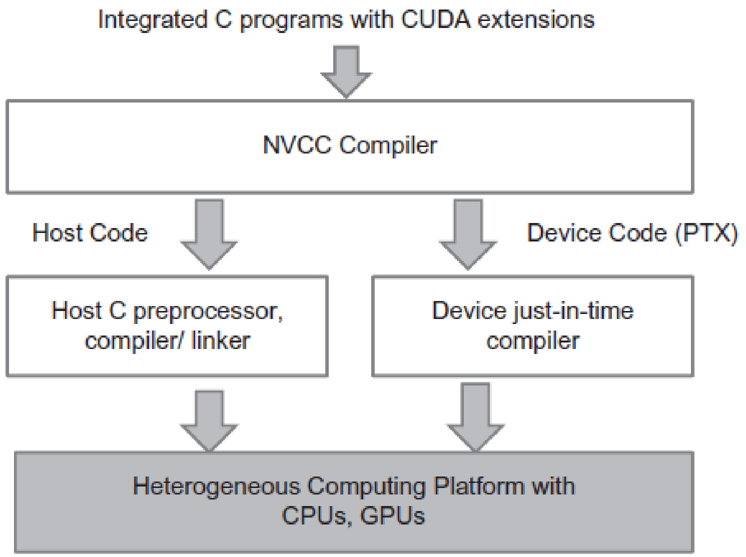
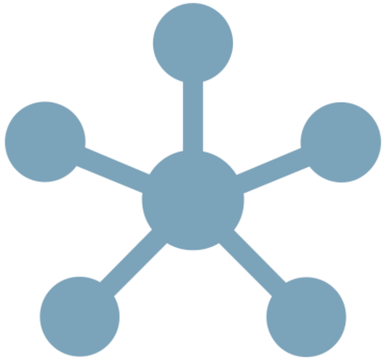


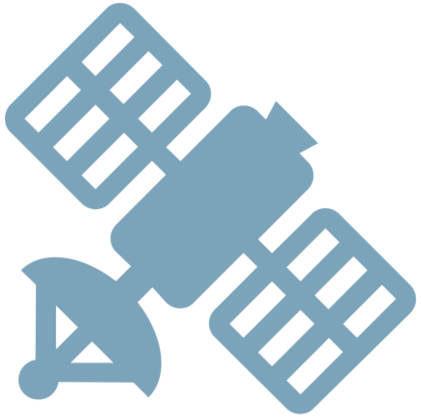
FIGURE 2.3

Overview of the compilation process of a CUDA C Program.

DEBUGGER: GDB (GNU) FOR CPU BASED



Network

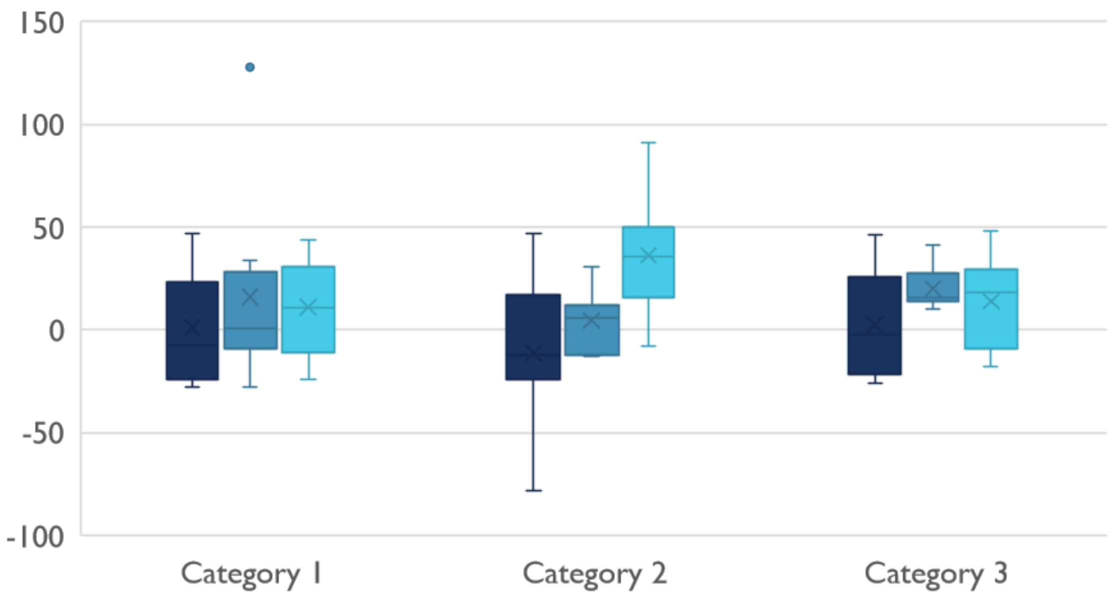


Satellite

Link

TECH REQUIREMENTS

COMPETITIVE LANDSCAPE



DIGITAL COMMUNICATIONS

- Cloud
- Local
- Hybrid

THANK YOU

TRAINING.PRAVEEN@GMAIL.COM