# 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

- CUDA theory
- Debugger: GDB (GNU) for CPU based
- Python C Profilers
- CPU Thread & Memory Hierarchy
- NVIDIA GPGPU Architecture
- GPU Thread & Memory Hierarchy
- GPGPU computing with CUDA
- CUDA constructs
- Profiling: GNU (gprof), Intel VTune
- Debugger: GDB (GNU) for GPU

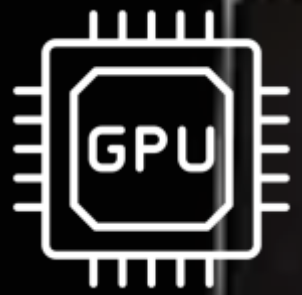

# INTRODUCTION TO CUDA

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.

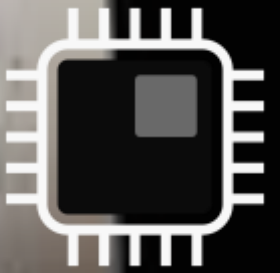
# But Why GPUs for Computation?




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

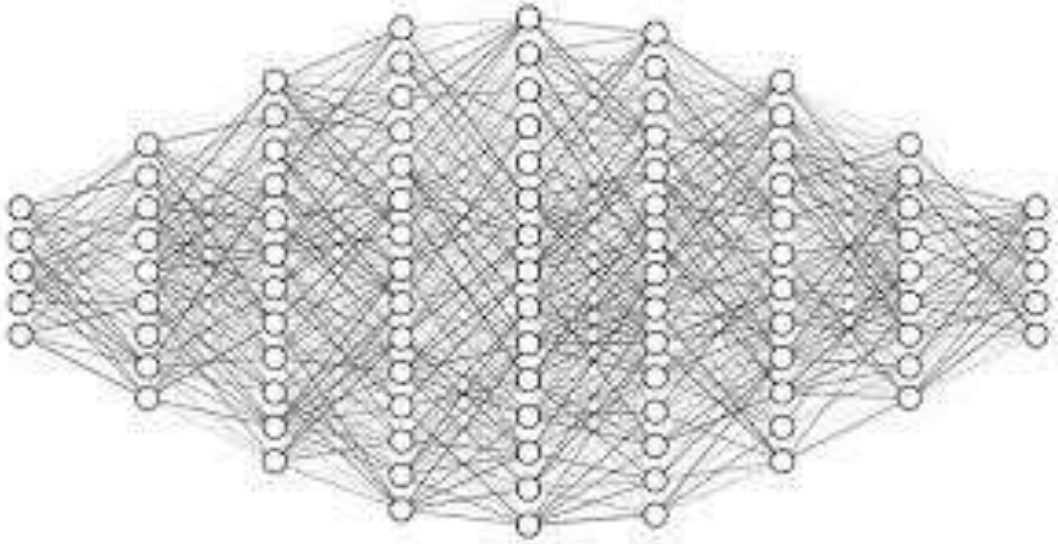
GPUs excel at processing large data sets in parallel.


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.

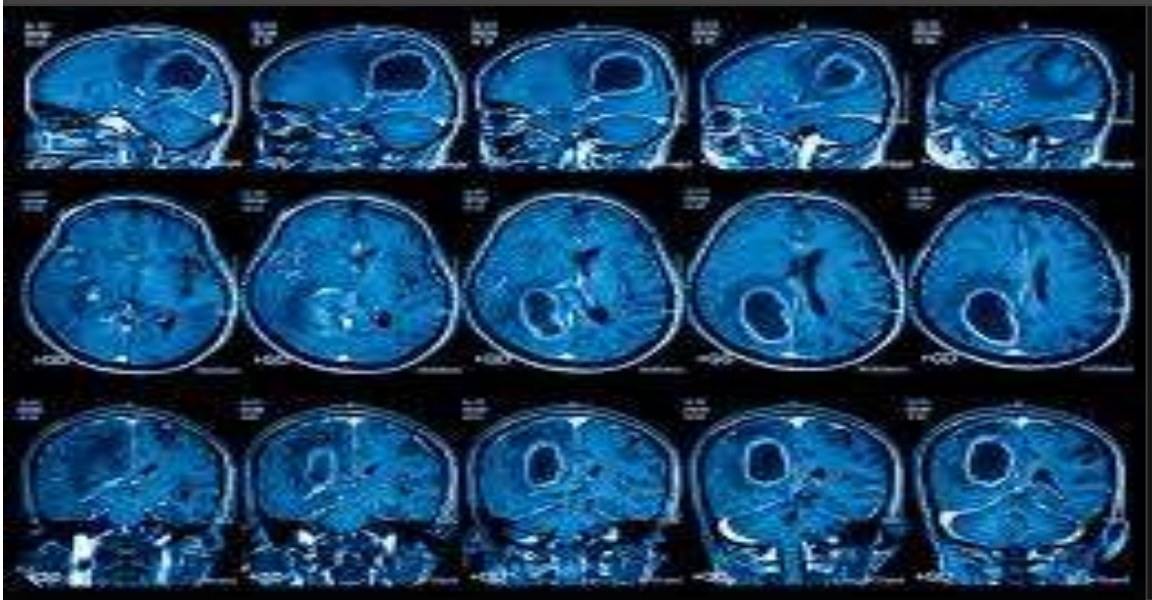


# REAL WORLD APPLICATIONS OF GPUs

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

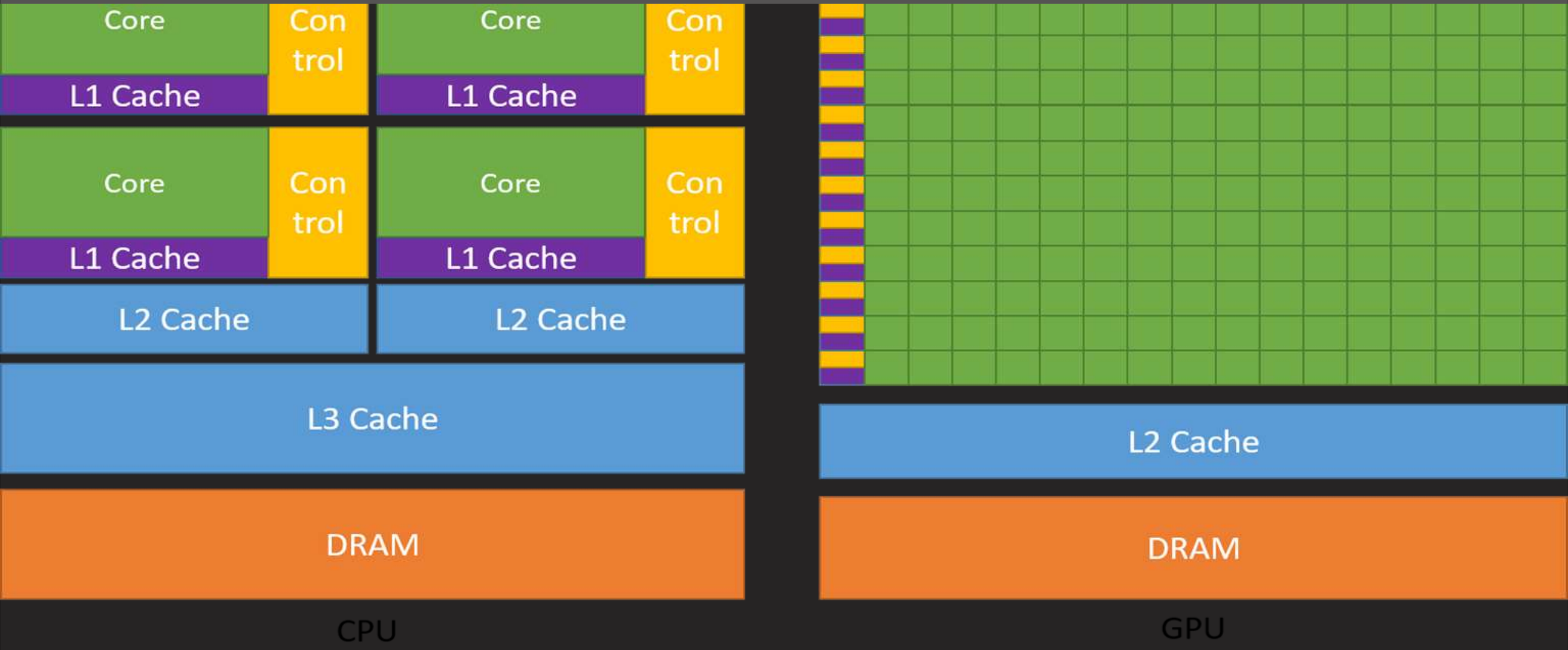
Scientists rely on GPUs for molecular modeling and running weather simulations faster than ever. GPU acceleration allows them to analyze complex systems with greater accuracy.

# REAL WORLD APPLICATIONS OF GPUs

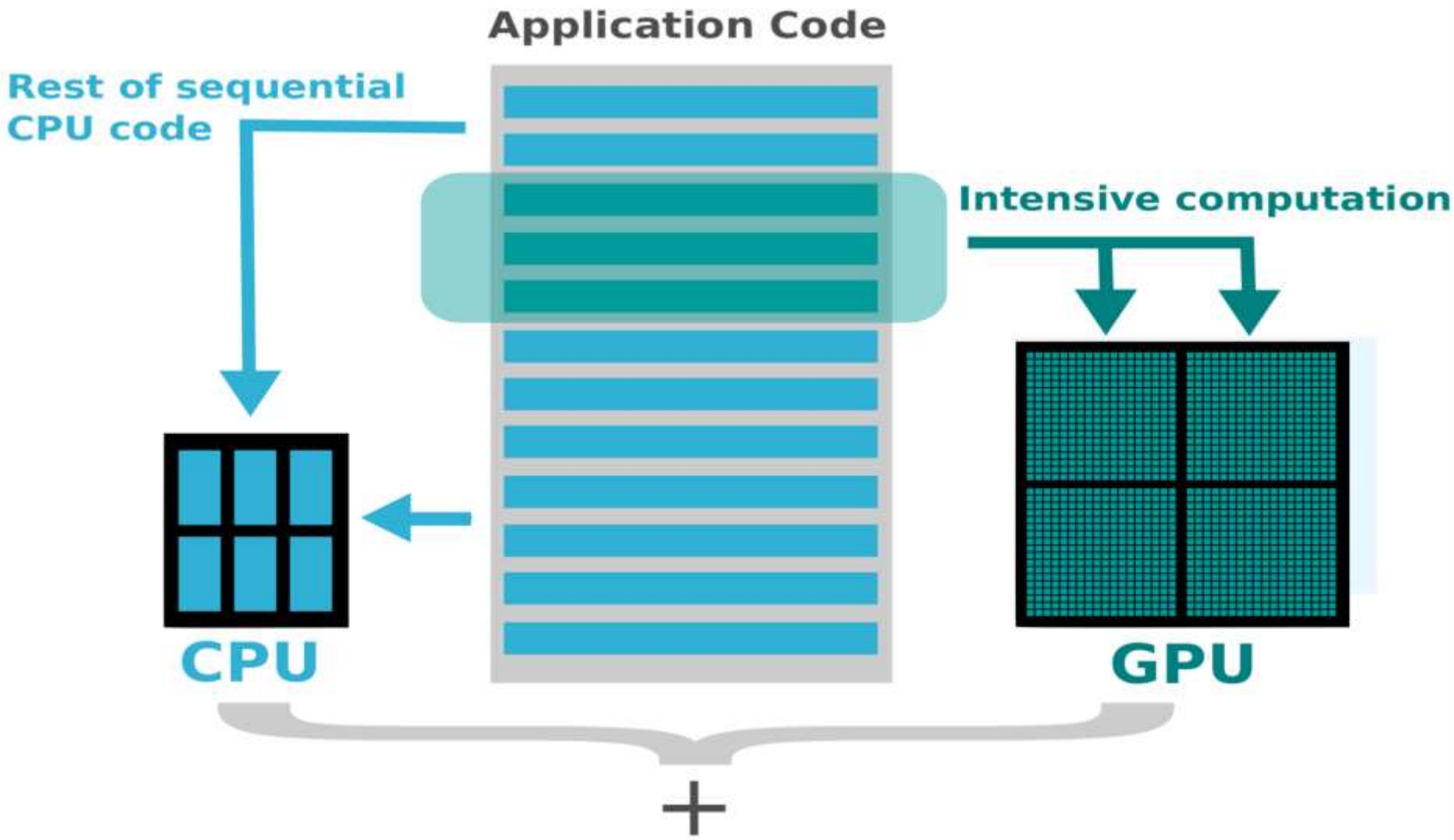


Doctors process and analyze medical images using GPUs to detect health issues more rapidly. Fast GPU performance means quicker diagnoses and better treatment decisions.

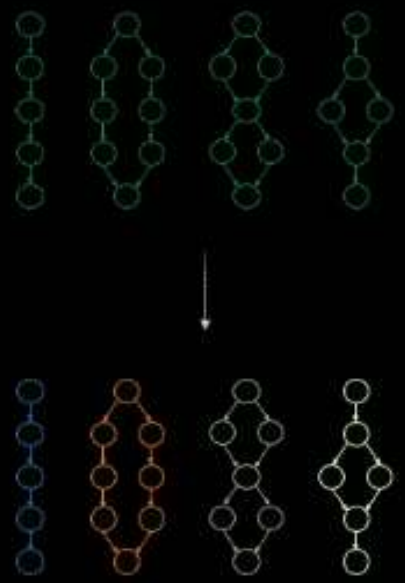
# CPU vs GPU ARCHITECTURE



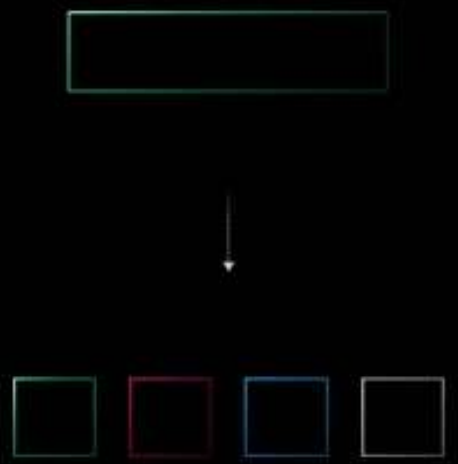
# HOW GPU ACCELERATION ACTUALLY WORKS?



# There Are Really Only Two Types of Parallelism Patterns



**Task Parallelism**
Divide independent **programs** across processors



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

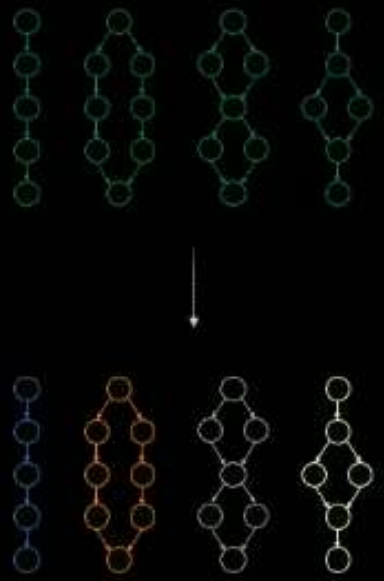
Task Parallelism

- 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

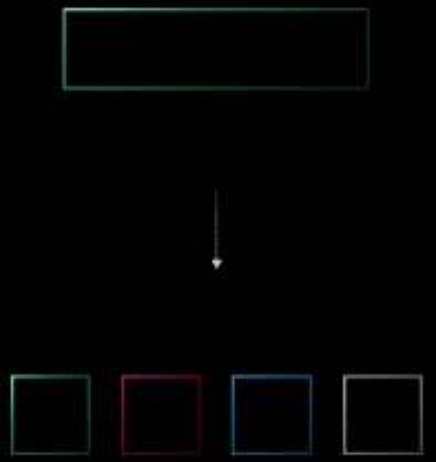
- A single program running across multiple threads/processors
- All threads execute the same operation on different elements of data in lockstep

# CUDA is Both: Data Parallelism Inside Task Parallelism



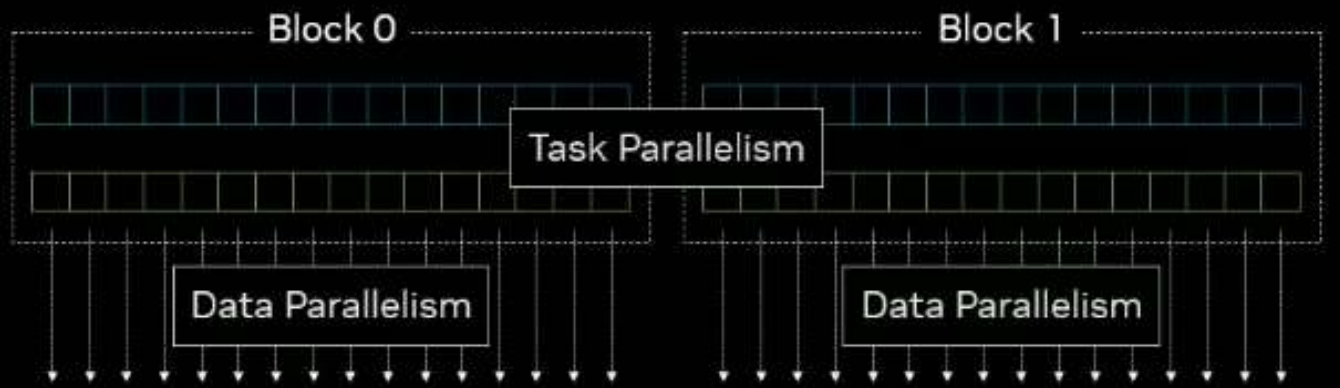
**Task Parallelism**

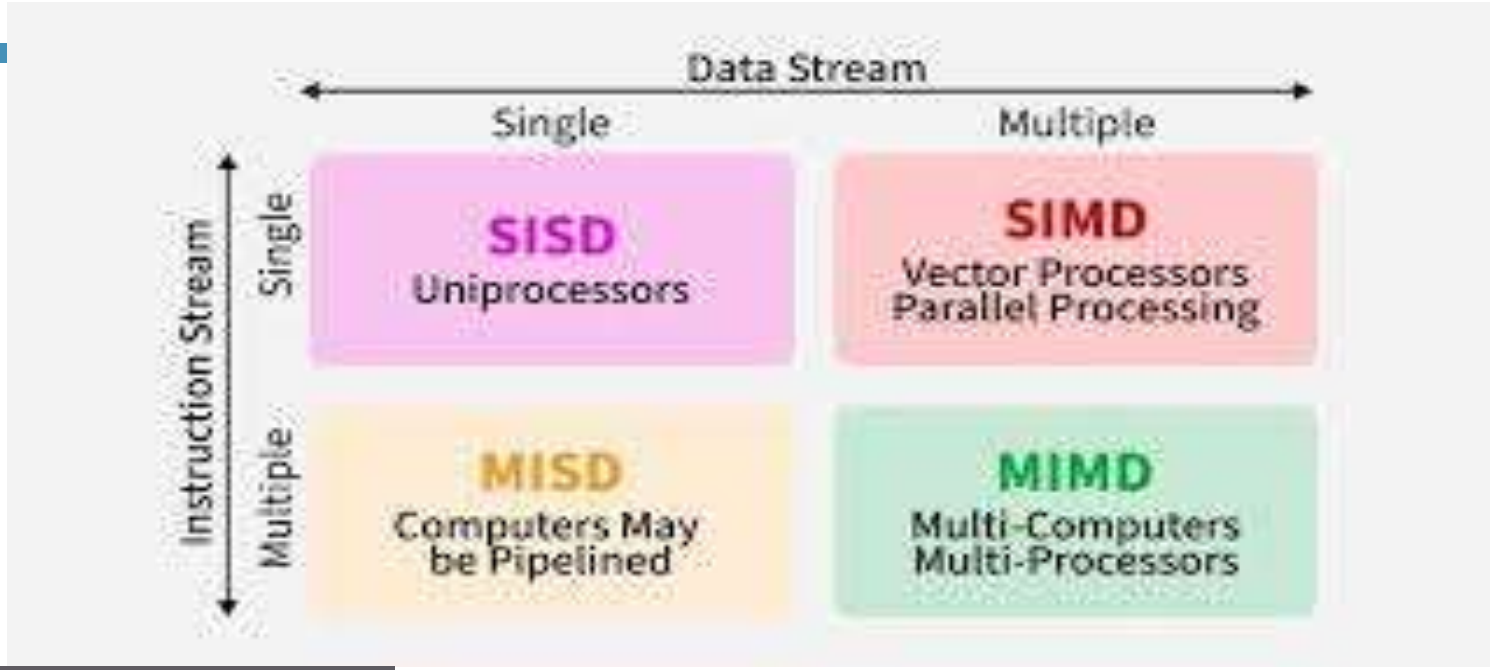
Divide independent **programs** across processors

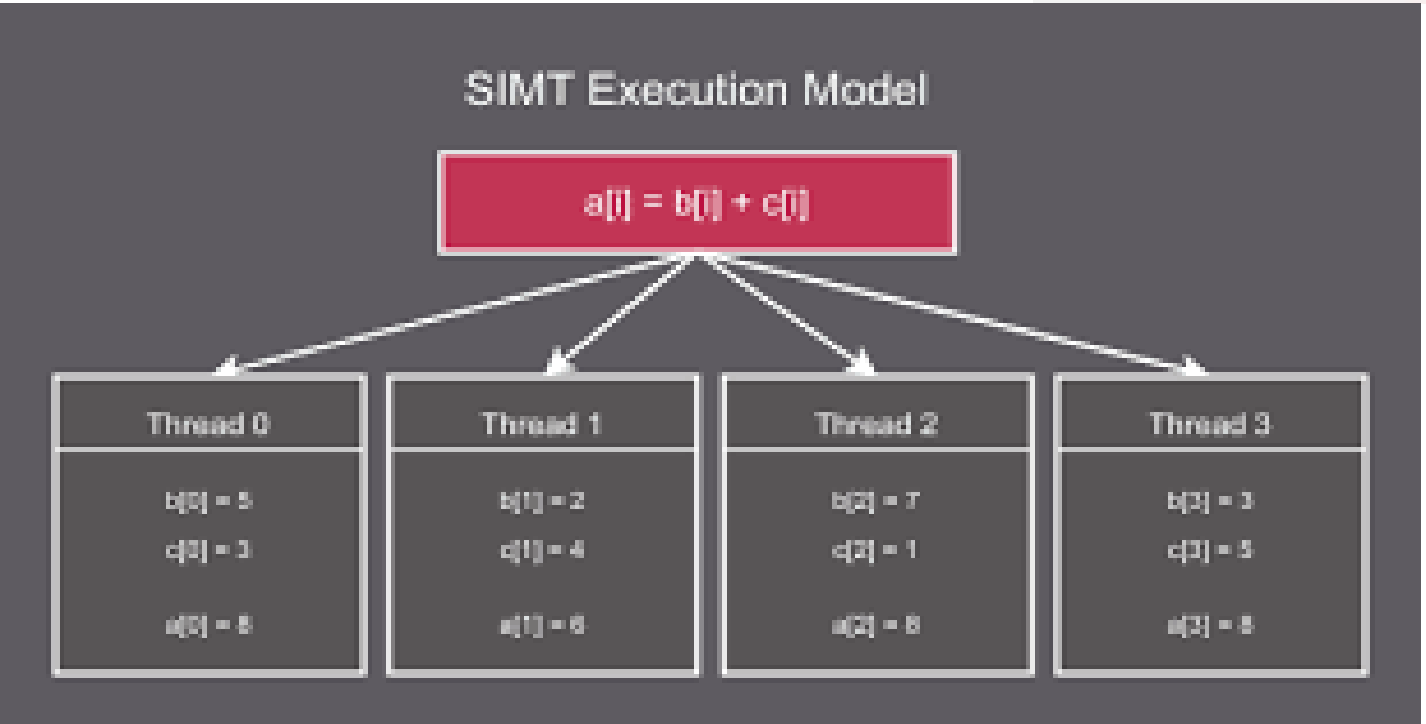


**Data Parallelism**

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

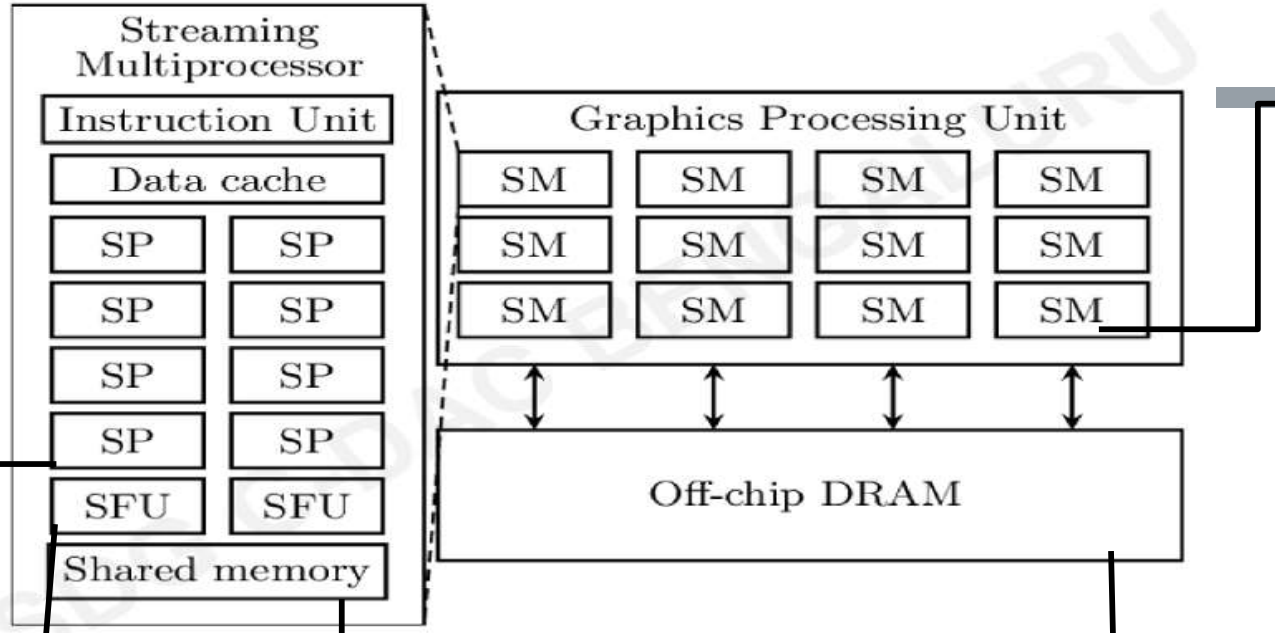

SIMT Execution Model
a[i] = b[i] + c[i]
Thread 0
b[0] = 5
c[0] = 3
a[0] = 8
Thread 1
b[1] = 2
c[1] = 4
a[1] = 6
Thread 2
b[2] = 7
c[2] = 1
a[2] = 8
Thread 3
b[3] = 3
c[3] = 5
a[3] = 8

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




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

Streaming Multiprocessor

-executes multiple threads in parallel

-Uses SIMT model

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

# 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

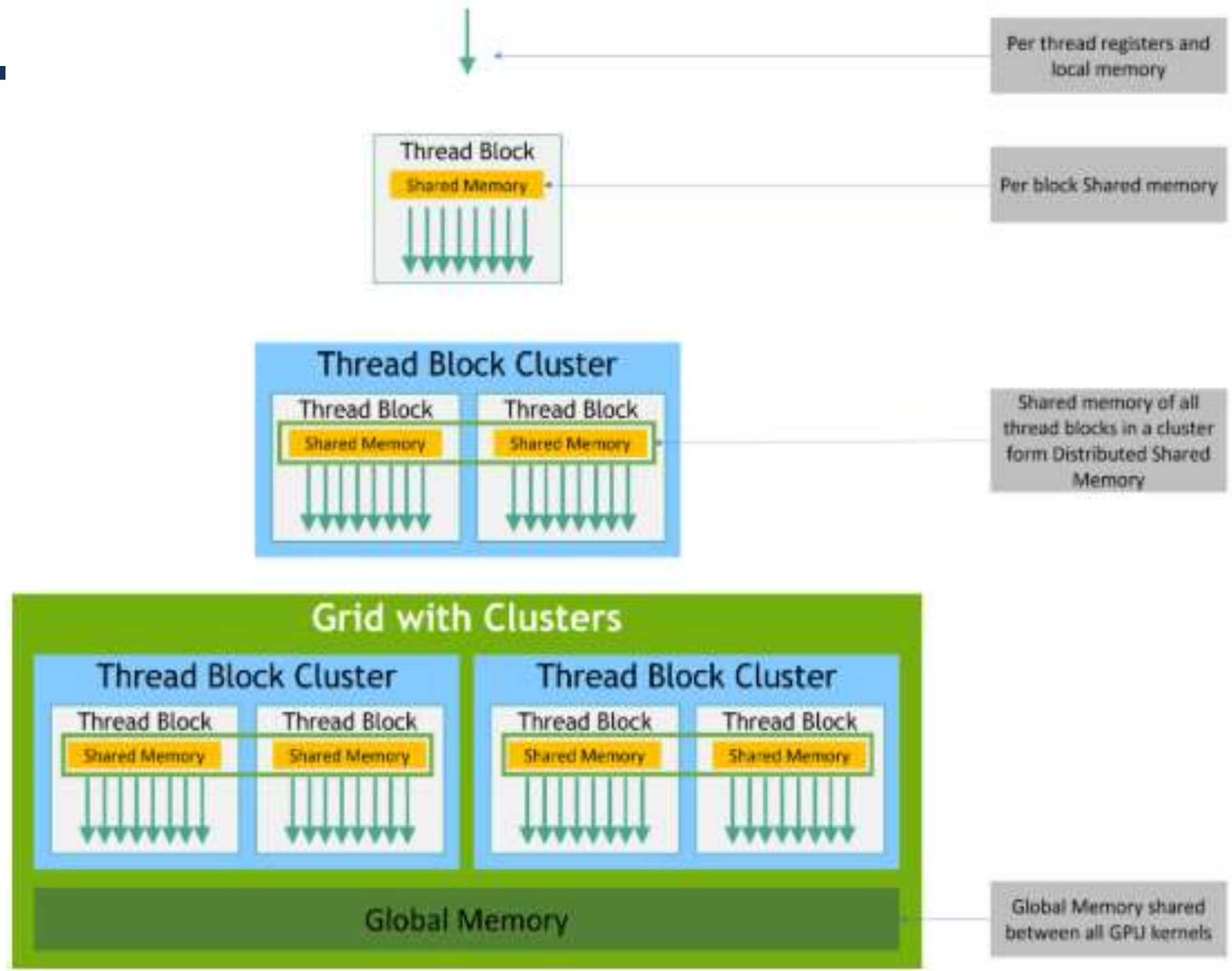
**CUDA development requires two compilers: one for the CPU (host) code and one for the GPU (device) code.**

**To compile CUDA code:**

- cmd> nvcc demo.cu -o demo

**To Run CUDA code:**

- cmd> ./demo

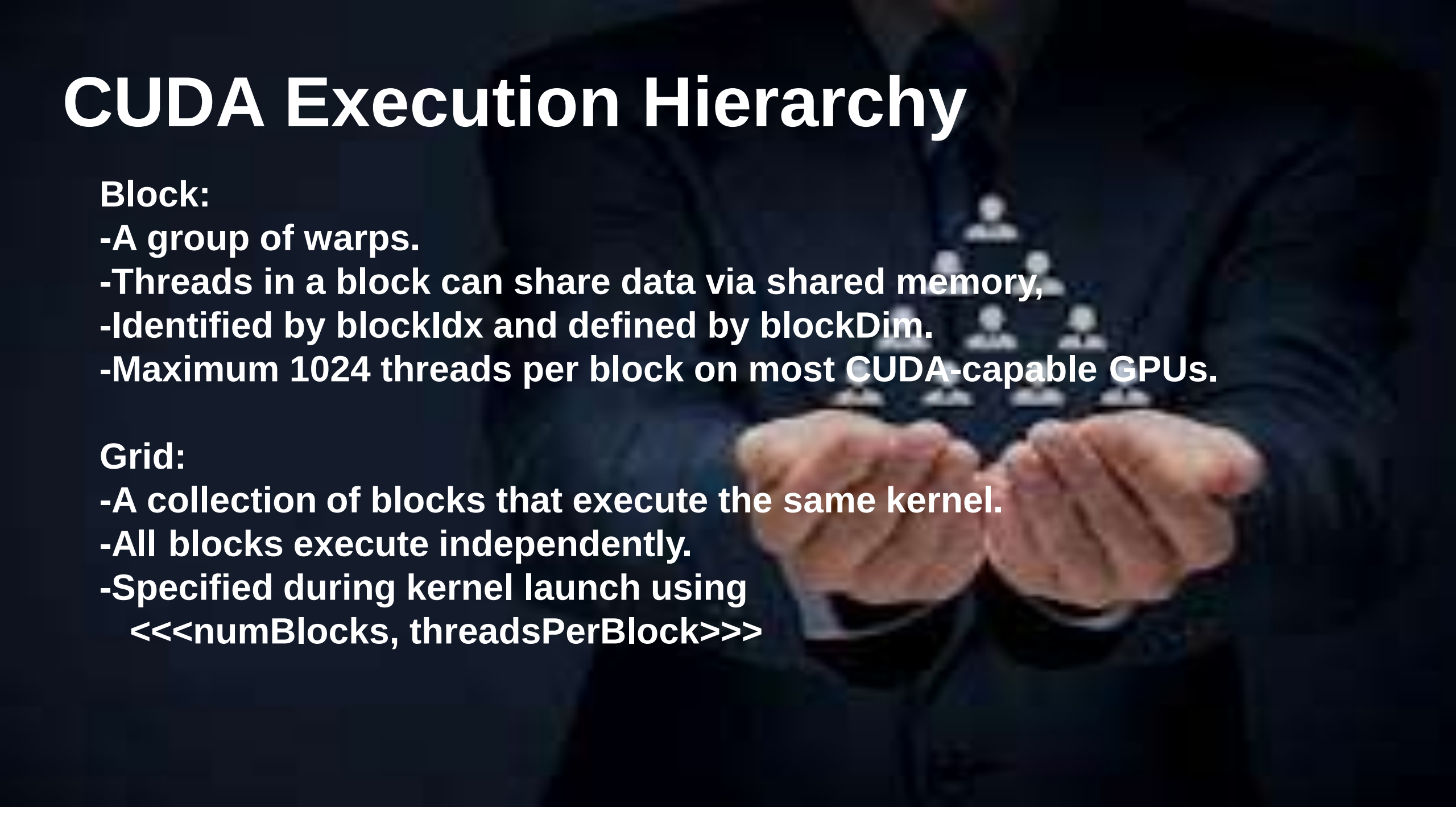

Per thread registers and local memory
Thread Block
Shared Memory
Per block Shared memory
Thread Block Cluster
Thread Block
Shared Memory
Thread Block
Shared Memory
Shared memory of all thread blocks in a cluster form Distributed Shared Memory
Grid with Clusters
Thread Block Cluster
Thread Block
Shared 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.**

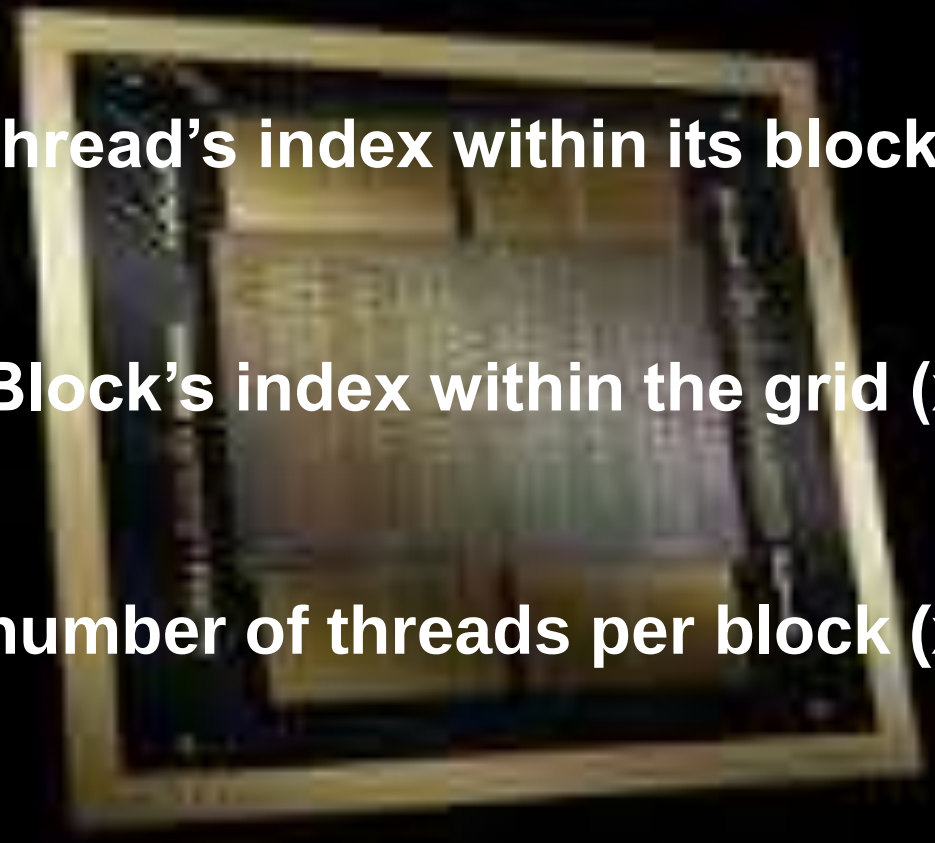
**Grid:**
**-A collection of blocks that execute the same kernel.**
**-All blocks execute independently.**
**-Specified during kernel launch using**
**<<<numBlocks, threadsPerBlock>>>**

# 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

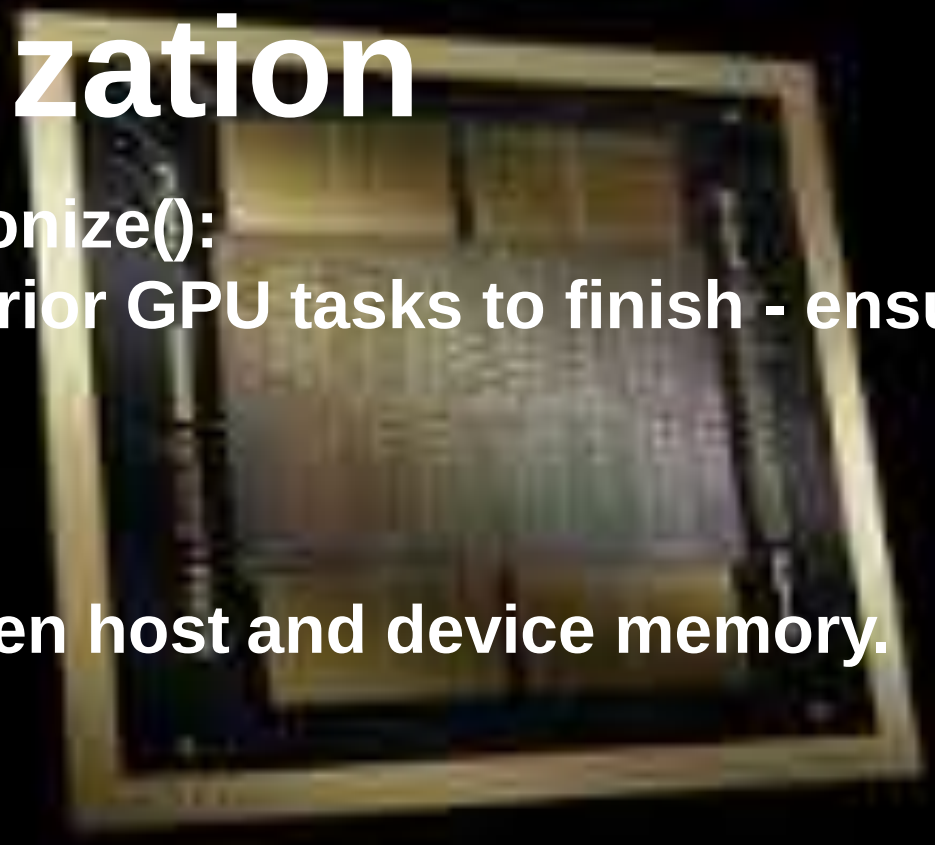


**threadIdx.x:**
**-Built-in variable: thread's index within its block (x-dimension).**

**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

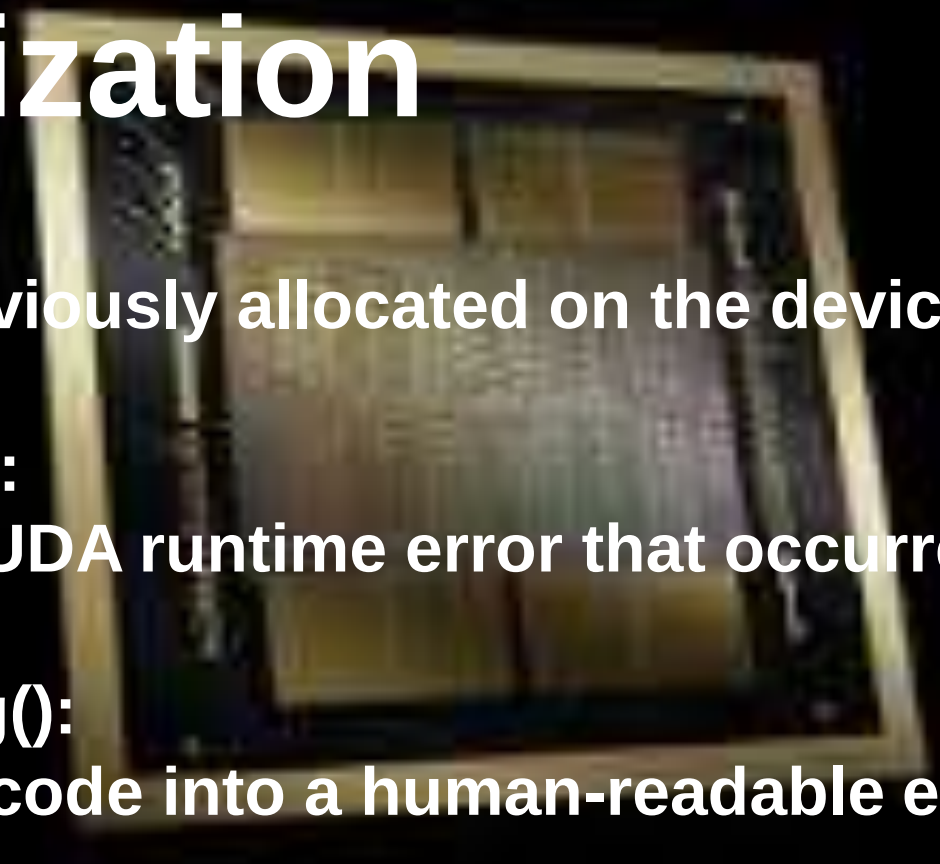


**cudaDeviceSynchronize():**
**-Host waits for all prior GPU tasks to finish - ensures result completion.**

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

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

# CUDA Memory and Synchronization

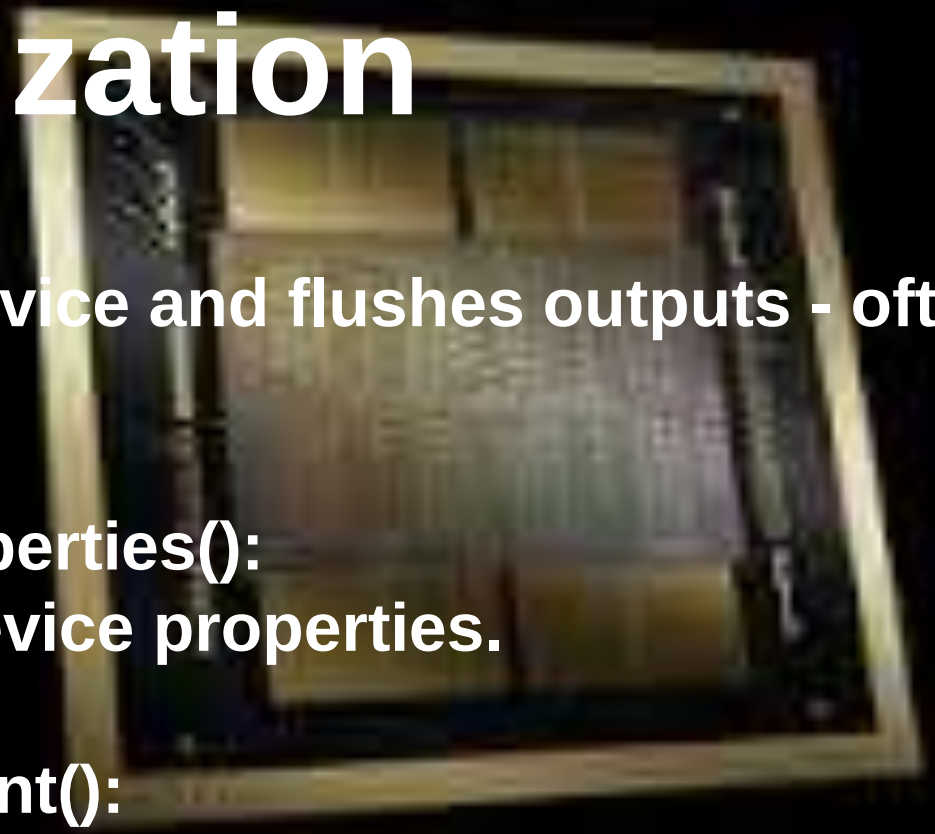


**cudaFree():**
**-Frees memory previously allocated on the device.**

**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!
Hello World from GPU!
(qdenoise_gpu) shreya@user:~$
```

THE CODE

```cpp
#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

```cpp
#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;
}
```

# OUTPUT

```
(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
```

```cpp
#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.

# DEBUGGER: GDB (GNU) FOR CPU BASED

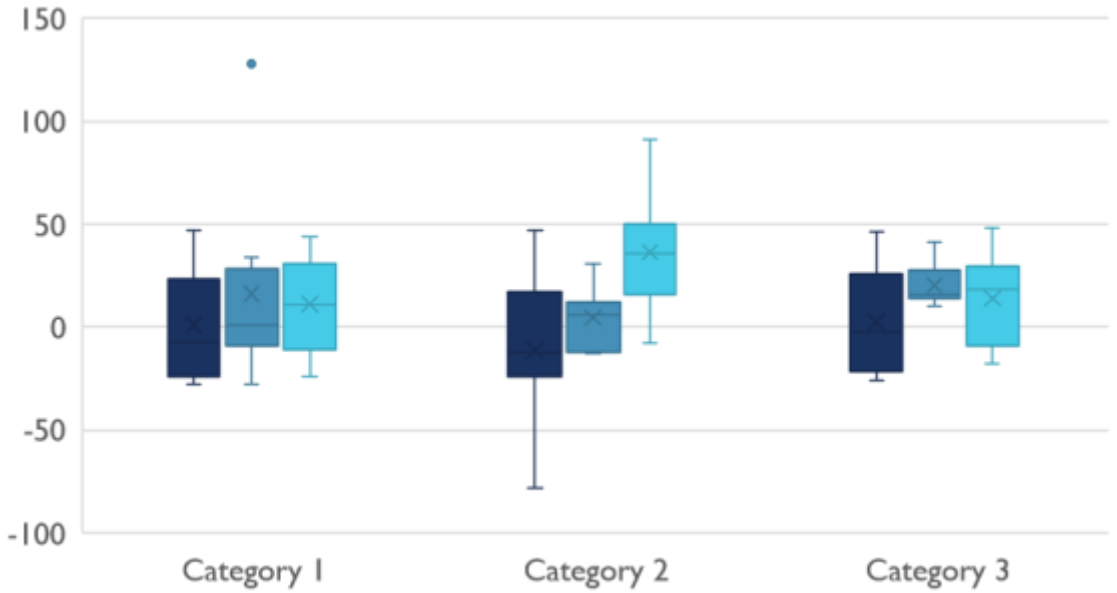
Network

Satellite

Link

# TECH REQUIREMENTS

# COMPETITIVE LANDSCAPE

# DIGITAL COMMUNICATIONS

- Cloud
- Local
- Hybrid

# THANK YOU

TRAINING.PRAVEEN@GMAIL.COM