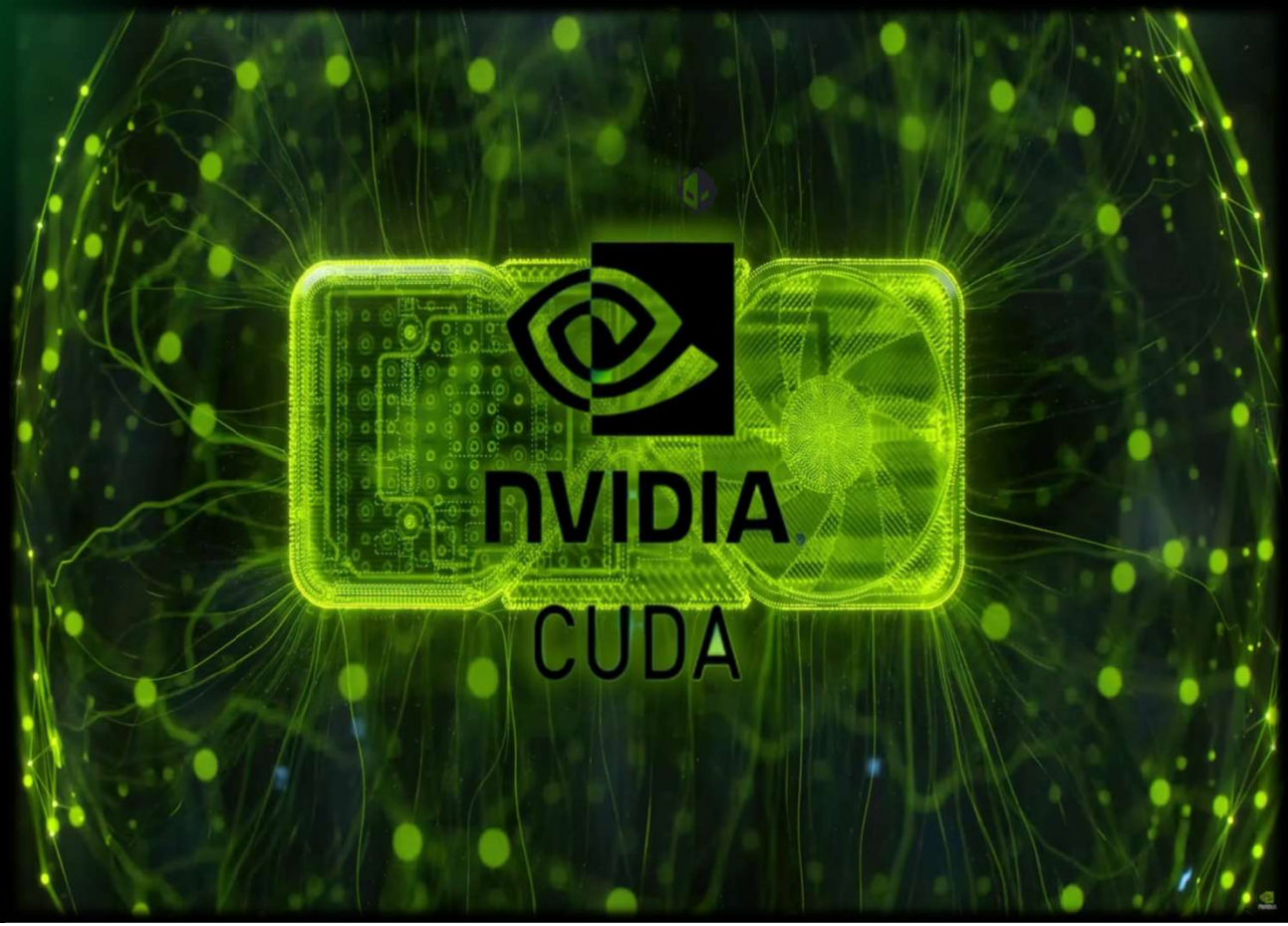
MIT WPU

# GPU PROGRAMMING & HPC OPTIMIZATION :

## CUDA

Done by : Abhas Jaltare
PRN - 1032221128



# INTRODUCTION

- GPU (Graphics Processing Unit) programming involves using specialized hardware to perform highly parallel computations. Unlike CPUs, which handle a few complex tasks at a time, GPUs execute thousands of smaller tasks simultaneously, making them ideal for high-performance computing (HPC).

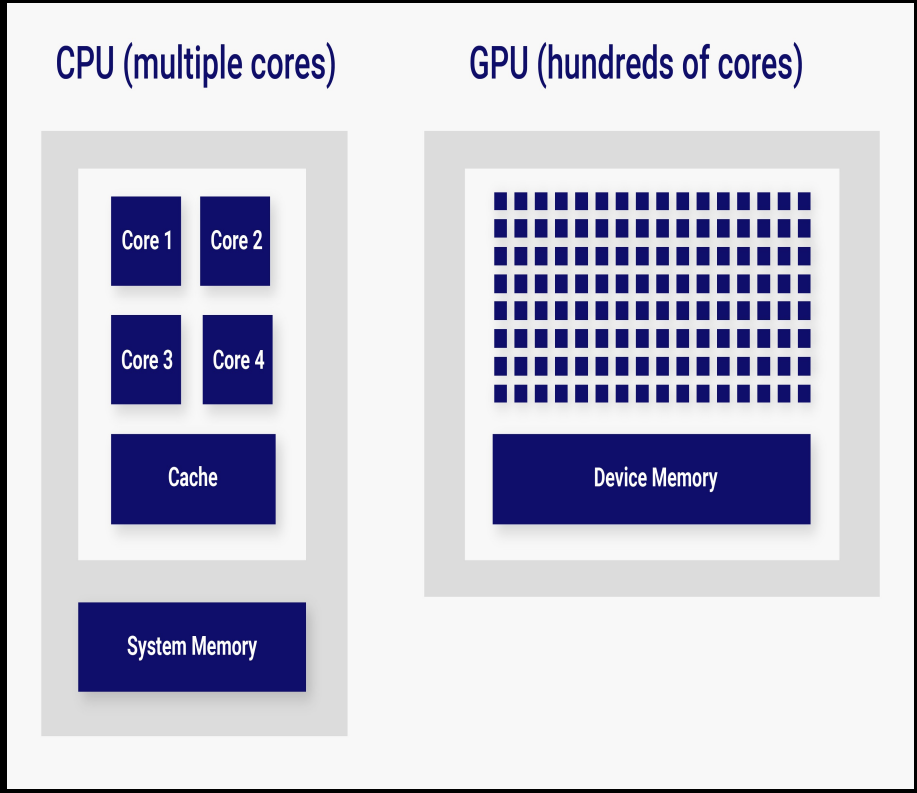
- HPC optimization ensures that computations run efficiently, reducing execution time and resource usage. It plays a crucial role in scientific simulations, artificial intelligence, and real-time processing.

- **CUDA (Compute Unified Device Architecture)** is a parallel computing platform developed by NVIDIA. It provides a programming model and toolkit that allows developers to harness the power of GPUs for general-purpose computing. With CUDA, tasks can be distributed across multiple GPU cores, significantly accelerating computation compared to traditional CPU-based processing.

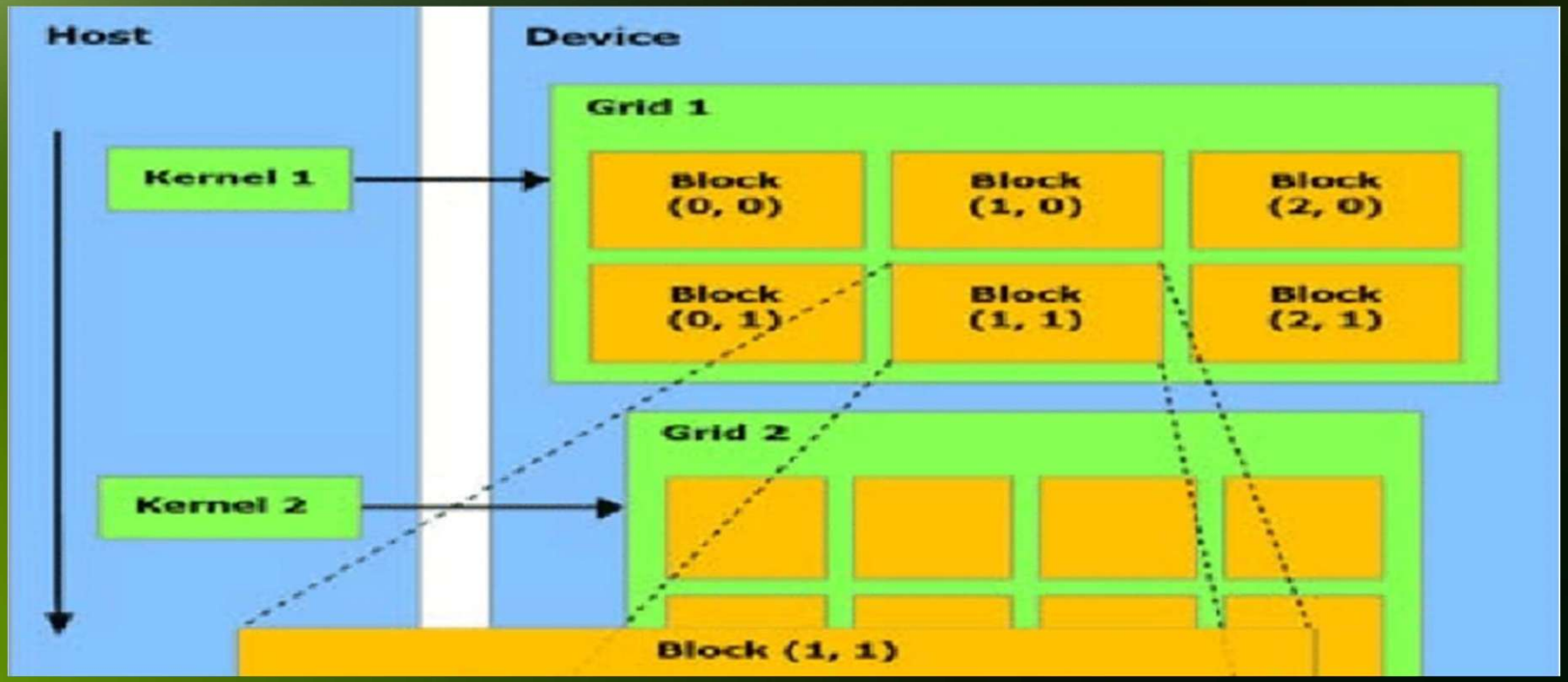
# BACKGROUND / MOTIVATION

Traditional **CPUs** are limited in handling highly parallel tasks, leading to performance bottlenecks.
**GPUs** enable massive parallelism, making them ideal for High-Performance Computing (HPC) applications.
**HPC Optimization** is crucial for improving speed, efficiency, and resource utilization in large-scale computations.

**CUDA**, developed by NVIDIA, simplifies GPU programming and accelerates computations in AI, scientific simulations, and real-time processing.
Optimizing CUDA-based applications helps achieve **faster execution, lower power consumption, and better scalability**.



# ARCHITECTURE




- In CUDA, The CPU sends small programs called **kernels** to the **GPU** to do work.
- The GPU breaks this work into **grids**, each grid has many **blocks**, and each block has many **threads** (tiny tasks).
- This structure helps the GPU run thousands of tasks at the same time, super fast and in parallel!

# LITERATURE REVIEW

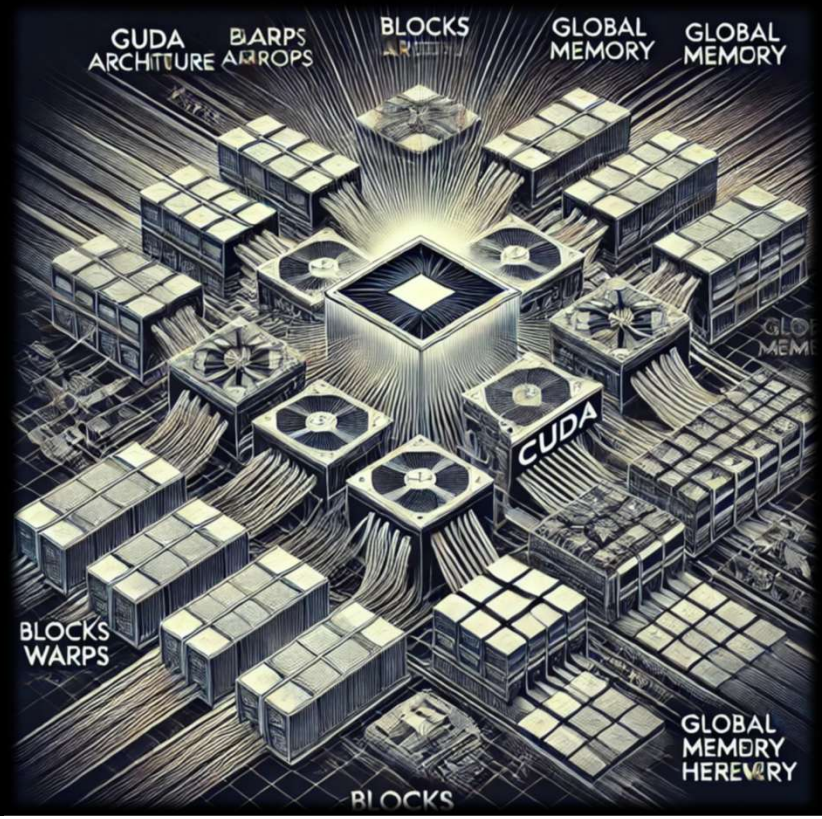
•**D. De Donno et al. (2010, IEEE)**: Explored GPU computing and CUDA programming, demonstrating performance improvements in electromagnetic simulations using the FDTD method.

•**J. Wu et al. (2019, IEEE)**: Developed a performance model for GPU architectures, optimizing on-chip resource allocation for medical image registration.

•**K. Zhou et al. (2022, IEEE)**: Introduced an automated tool for analyzing and tuning GPU-accelerated HPC applications, enhancing performance efficiency.

•**J. Nickolls (2007, IEEE)**: Provided foundational insights into CUDA's parallel computing architecture, highlighting its advantages over traditional CPU-based processing.

•**A. Asaduzzaman et al. (2021, IEEE)**: Compared CUDA and OpenCL in parallel and distributed computing, evaluating performance trade-offs and energy efficiency.



1. The evolution of GPU programming has significantly influenced high-performance computing (HPC), with CUDA emerging as a key technology in this field. **De Donno et al. (2010)** provided an early exploration of CUDA programming, demonstrating its effectiveness in accelerating electromagnetic simulations using the Finite-Difference Time-Domain (FDTD) method. Their work highlighted the potential of CUDA in scientific computing by leveraging parallel execution for improved performance.

2. Expanding on GPU performance modeling, **Wu et al. (2019)** introduced a resource-aware model for GPU architectures, particularly applied to medical image registration. Their research addressed optimization challenges related to on-chip resource allocation, demonstrating how efficient memory management can enhance computational speed and accuracy in medical applications.

3. A more recent study by **Zhou et al. (2022)** focused on automated tools for analyzing and optimizing GPU-accelerated HPC applications. Their work emphasized the need for systematic performance tuning techniques, which help developers maximize computational efficiency in large-scale parallel applications.

4. From an architectural perspective, **Nickolls (2007)** provided foundational insights into CUDA's programming model, illustrating how its parallel computing architecture differs from traditional CPU-based processing. This research laid the groundwork for CUDA's widespread adoption in fields such as AI, machine learning, and real-time simulations.

5. Finally, **Asaduzzaman et al. (2021)** compared CUDA with OpenCL in parallel and distributed computing environments. Their study analyzed performance trade-offs, energy efficiency, and ease of implementation, concluding that CUDA offers superior optimization capabilities but remains limited to NVIDIA hardware. Collectively, these studies underscore CUDA's role in revolutionizing GPU computing, optimizing memory usage, and enhancing parallel processing for various HPC applications. However, challenges remain in multi-GPU scaling, debugging, and portability, indicating the need for further research in these areas.

# DEMONSTRATION

smaller_data.cu > ...

```cpp
#include <iostream>
#include <chrono>
#include <cuda_runtime.h>

__global__ void gpu_add(int* d_a, int* d_b, int* d_c, int n) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) d_c[idx] = d_a[idx] + d_b[idx];
}

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

int main() {
    const int N = 1 << 20; // 1 million elements
    size_t size = N * sizeof(int);

    int *h_a = new int[N];
    int *h_b = new int[N];
    int *h_c = new int[N];
    int *h_c_gpu = new int[N];

    for (int i = 0; i < N; ++i) {
        h_a[i] = rand() % 100;
        h_b[i] = rand() % 100;
    }

    // CPU addition timing
    auto start_cpu = std::chrono::high_resolution_clock::now();
    cpu_add(h_a, h_b, h_c, N);
    auto end_cpu = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> cpu_time = end_cpu - start_cpu;

    // GPU memory allocation and copy
    int *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);

    // GPU kernel timing
    auto start_gpu = std::chrono::high_resolution_clock::now();
    gpu_add<<<(N + 255) / 256, 256>>>(d_a, d_b, d_c, N);
```

## TESTING FOR SMALL – MODERATE DATA

```cpp
    cudaDeviceSynchronize();
    auto end_gpu = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> gpu_time = end_gpu - start_gpu;

    cudaMemcpy(h_c_gpu, d_c, size, cudaMemcpyDeviceToHost);

    // Compare CPU and GPU results
    bool match = true;
    for (int i = 0; i < N; ++i) {
        if (h_c[i] != h_c_gpu[i]) {
            match = false;
            break;
        }
    }

    std::cout << "CPU Time: " << cpu_time.count() << " s\n";
    std::cout << "GPU Time: " << gpu_time.count() << " s\n";
    std::cout << "Results match: " << (match ? "Yes" : "No") << "\n";

    // 🚀 Speedup output
    std::cout << "Speedup (CPU / GPU): " << (cpu_time.count() / gpu_time.count()) << "x\n";

    // Cleanup
    delete[] h_a;
    delete[] h_b;
    delete[] h_c;
    delete[] h_c_gpu;
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

# DEMONSTRATION

bigger_data.cu > ...

```cpp
#include <iostream>
#include <chrono>
#include <cuda_runtime.h>

__global__ void gpu_add(int* d_a, int* d_b, int* d_c, int n) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) d_c[idx] = d_a[idx] + d_b[idx];
}

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

int main() {
    const int N = 1 << 25; // 33,554,432 elements (approx 33 million)
    size_t size = N * sizeof(int);

    int *h_a = new int[N];
    int *h_b = new int[N];
    int *h_c = new int[N];
    int *h_c_gpu = new int[N];

    for (int i = 0; i < N; ++i) {
        h_a[i] = rand() % 100;
        h_b[i] = rand() % 100;
    }

    // CPU addition timing
    auto start_cpu = std::chrono::high_resolution_clock::now();
    cpu_add(h_a, h_b, h_c, N);
    auto end_cpu = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> cpu_time = end_cpu - start_cpu;

    // GPU memory allocation and copy
    int *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);

    // GPU kernel timing
    auto start_gpu = std::chrono::high_resolution_clock::now();
    gpu_add<<<(N + 255) / 256, 256>>>(d_a, d_b, d_c, N);
```

## TESTING FOR - LARGE DATA

```cpp
    cudaDeviceSynchronize();
    auto end_gpu = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> gpu_time = end_gpu - start_gpu;

    cudaMemcpy(h_c_gpu, d_c, size, cudaMemcpyDeviceToHost);

    // Compare CPU and GPU results
    bool match = true;
    for (int i = 0; i < N; ++i) {
        if (h_c[i] != h_c_gpu[i]) {
            match = false;
            break;
        }
    }

    std::cout << "CPU Time: " << cpu_time.count() << " s\n";
    std::cout << "GPU Time: " << gpu_time.count() << " s\n";
    std::cout << "Results match: " << (match ? "Yes" : "No") << "\n";
    std::cout << "Speedup (CPU / GPU): " << (cpu_time.count() / gpu_time.count()) << "x\n";

    // Cleanup
    delete[] h_a;
    delete[] h_b;
    delete[] h_c;
    delete[] h_c_gpu;
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

# EXECUTION RESULTS

## SMALL-MODERATE DATA :

```
C:\CUDAPROJ>nvcc -arch=sm_75 smaller_data.cu -o smaller_data.exe
smaller_data.cu
tmpxft_00003244_00000000-7_smaller_data.cudafe1.cpp
   Creating library smaller_data.lib and object smaller_data.exp

C:\CUDAPROJ>smaller_data.exe
CPU Time: 0.0026614 s
GPU Time: 0.0028139 s
Results match: Yes
Speedup (CPU / GPU): 0.945805x
```

🔹 **1. Smaller Dataset**

•**CPU Time:** ~ 2.6 ms

•**GPU Time:** ~ 2.8 ms

•**Speedup: < 1** → CPU was slightly faster

•For small datasets, the **overhead** of launching GPU kernels and copying data **outweighs the benefit** of parallelism. The CPU can handle small operations **in-place and with fewer steps**, so it's often faster for such tasks.

## LARGE DATA :

```
C:\CUDAPROJ>nvcc -arch=sm_75 bigger_data.cu -o bigger_data.exe
bigger_data.cu
tmpxft_00006418_00000000-7_bigger_data.cudafe1.cpp
   Creating library bigger_data.lib and object bigger_data.exp

C:\CUDAPROJ>bigger_data.exe
CPU Time: 0.087334 s
GPU Time: 0.0029985 s
Results match: Yes
Speedup (CPU / GPU): 29.1259x
```

🔹 **2. Bigger Dataset**

•**CPU Time:** ~ 87.3 ms

•**GPU Time:** ~ 3.0 ms

•**Speedup: 29.12x**

•With **larger datasets**, the GPU's thousands of cores can operate in **parallel**, making it **much more efficient**. This is the **ideal scenario for GPU acceleration**: high-volume, parallelizable tasks.

# RESEARCH GAP

**POWER CONSUMPTION**

GPUs consume high power, requiring energy-efficient optimization strategies.

**DEBUGGING & PROFILING ISSUES**

Lack of easy-to-use debugging tools for optimizing CUDA applications.

**LIMITED MULTI GPU SUPPORT**

Challenges in efficiently distributing tasks across multiple GPUs.

**REAL TIME PROCESSING CHALLENGES**

Improving CUDA's performance for time-sensitive applications like robotics and finance.

**STEEP LEARNING CURVE**

CUDA programming requires advanced knowledge, making it difficult for beginners.

# Pros & Cons :

| PROS | CONS |
| --- | --- |
| **High Performance** → Enables massive parallelism, significantly speeding up computations. | **Limited to NVIDIA GPUs** → CUDA is proprietary and does not work on AMD or other GPUs. |
| **Energy Efficient** → Executes tasks faster than CPUs, reducing power consumption per computation. | **Steep Learning Curve** → CUDA programming requires understanding specific concepts like threads, blocks, and memory management on GPUs, which can be challenging for developers unfamiliar with parallel computing. |
| **Rich Ecosystem** → Provides extensive libraries (cuDNN, TensorRT) for scientific and AI applications. | **High Power Consumption** → GPUs require significant power, leading to higher operational costs. |
| **Large Community and Support** → Due to its widespread adoption, CUDA has a large community of developers and readily available support resources | **Debugging Complexity** → Debugging CUDA applications is more challenging than CPU-based programs. |
| **Unified Memory** → CUDA's unified memory feature allows seamless data transfer between CPU and GPU, simplifying memory management. | **Memory Constraints** → GPUs have limited memory compared to CPUs, usually known as VRAM (VIDEO RAM) , therefore careful optimization must be done . |

# APPLICATIONS

**Image processing** → CUDA is used for image editing, medical imagery, and seismic imaging.

**Gaming & Graphics Rendering** → Powers real-time ray tracing and high-performance graphics in modern video games.

**Finance & High-Frequency Trading** → Speeds up risk analysis, fraud detection, and real-time stock market predictions.

**Machine learning** → CUDA accelerates the training of models and speeds up inference.

# CONCLUSION & REFERENCES

CUDA has revolutionized GPU programming by enabling high-performance computing across various domains. By leveraging parallel processing and advanced optimization techniques, CUDA significantly enhances computational speed and efficiency compared to traditional CPU-based approaches. Its impact is evident in fields such as artificial intelligence, deep learning, scientific simulations, gaming, finance, and real-time data processing. Despite challenges like a steep learning curve and hardware limitations, CUDA remains the industry standard for GPU acceleration. As technology advances, improvements in multi-GPU computing, AI integration, and energy-efficient optimizations will further expand its applications. With its continuous evolution, CUDA is set to play a crucial role in shaping the future of high-performance computing, making complex tasks faster, more efficient, and more accessible than ever before.

**REFERENCES :**


https://www.nvidia.com/en-in/
https://openai.com/index/chatgpt/
https://www.youtube.com/

IEEE Papers: Focus on real-world applications of quantum computing in space missions, including material discovery, propulsion optimization, and deep-space communication.

ACM Papers: Highlight the development of quantum algorithms for trajectory mapping, spacecraft energy efficiency, and real-time space mission planning.

Springer Papers: Cover theoretical advancements in quantum mechanics, quantum error correction, and their potential role in interstellar exploration.