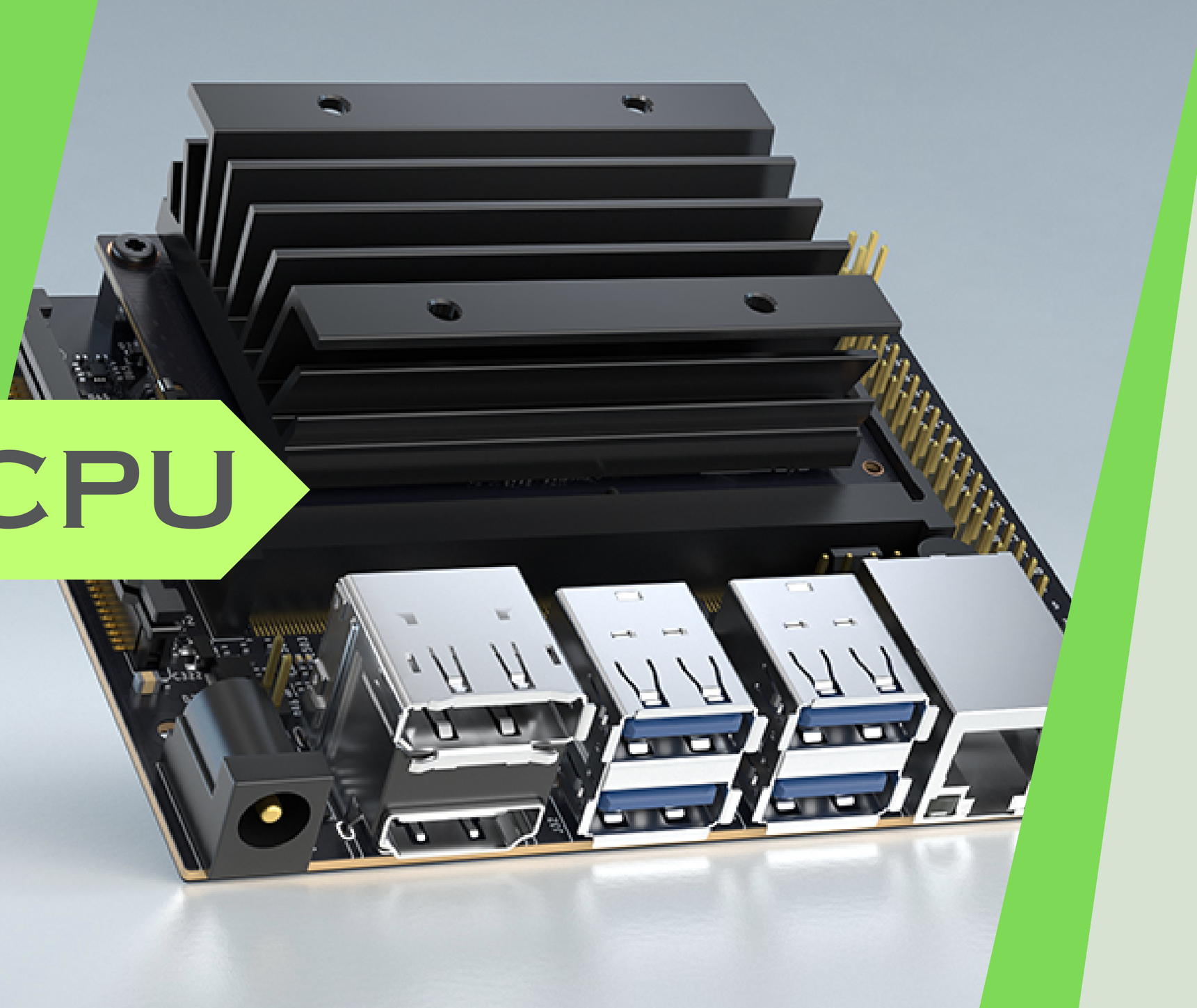
GPU vs CPU

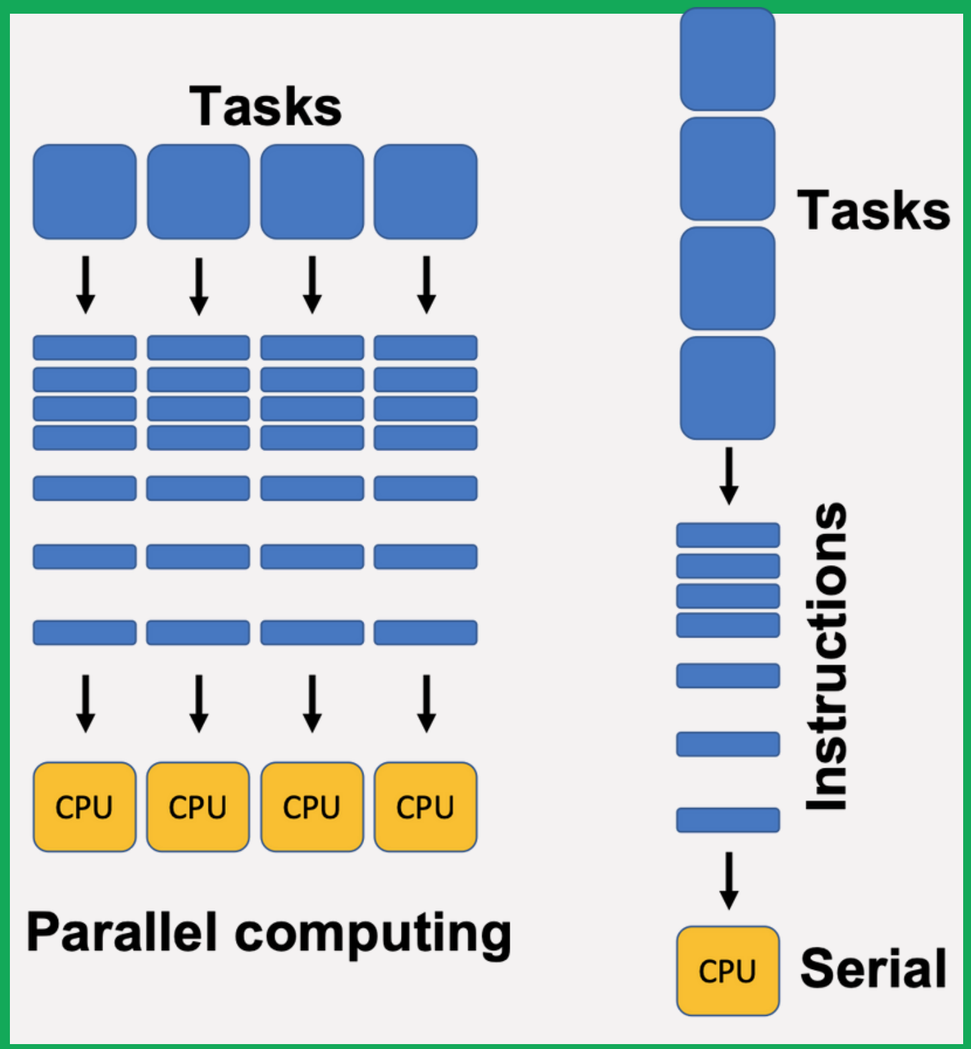
Advanced Algorithms performance comparison and parallel implementation



What is Parallel Programming?

Dividing a large task into smaller sub-tasks that can be processed simultaneously by multiple processors. This allows for faster processing times and increased efficiency, which is especially important in today's computing landscape where data sets are becoming larger and more complex.

Parallel Programming in CPU



Multithreading

is a common technique where a program is divided into multiple threads, each representing a separate sequence of instructions. These threads can be executed simultaneously on different CPU cores.

Task Parallelism

focuses on dividing a program into smaller tasks that can be executed independently. Each task represents a distinct unit of work that can be executed on different CPU cores.

Parallel Algorithms

Certain algorithms are inherently parallelizable, meaning they can be efficiently executed in parallel on CPUs. These algorithms leverage the ability to perform computations on different data elements or segments simultaneously.

Parallel Programming in GPU

Set up the development environment;

Install the necessary GPU drivers, SDK. CUDA is specific to NVIDIA GPUs, while OpenCL is more vendor-neutral and supports a wider range of GPU architectures.

Allocate and transfer data

Transfer data from the CPU's memory to the GPU's memory using data transfer APIs provided by the GPU framework.

Write GPU kernels

Kernels are functions or procedures written in CUDA or OpenCL that perform the actual computations on the GPU. Kernels are executed in parallel by multiple threads.

Launch GPU kernels

Invoke GPU kernels from the CPU code, specifying the number of threads or work items to execute.

Synchronize and retrieve results

Synchronize GPU execution to ensure all GPU kernels have been completed.

Cleanup and release resources

Free allocated GPU memory and release any resources acquired during the GPU computation.

CUDA

- **Streaming Multiprocessors (SMs):** These are the building blocks of the GPU and are responsible for executing parallel threads.
- **Host:** Refers to the CPU and its associated memory.
- **Device:** Refers to the GPU and its memory, where parallel computations are executed.
- **Threads:** Individual units of work that execute concurrently on the GPU.
- **Blocks:** Groups of threads that execute together and can synchronize and share data through shared memory.
- **Grids:** Collections of blocks that can execute independently.
- **Warps**: Groups of 32 threads that execute simultaneously on an SM.

CUDA programming model

- **Host Code:** Sequential code executed on the CPU, used for managing data and launching parallel tasks on the GPU.
- **Device Code:** Parallel code executed on the GPU, written in CUDA C/C++ and executed by thousands of threads.
- **Kernel:** A function that runs in parallel on the GPU and is invoked from the host code.
- **Thread Hierarchy:** Threads are organized into a hierarchy of blocks and grids for efficient parallel execution.
- **Thread Indexing:** Each thread is assigned a unique thread ID, which is used to access data and perform computations.
- **Memory Hierarchy:** Different memory types (global, shared, and local) are available for efficient data access and sharing.

CUDA API calls

- **cudaMalloc**: Allocates memory on the GPU.
- **cudaMemcpy**: Copies data between the host and device.
- **cudaMemcpyHostToDevice**: Copies data from host to device.
- **cudaMemcpyDeviceToHost**: Copies data from device to host.
- **cudaMemcpyDeviceToDevice**: Copies data between different locations on the device.
- **cudaFree**: Frees memory on the GPU.
- **cudaDeviceSynchronize**: Waits for the launched kernels on the current device to complete.

CUDA Simple vector add code

```
#include <stdio.h>

// CUDA kernel for vector addition
__global__ void vectorAdd(int *a, int *b, int *c, int n) {
    // Get the thread index
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    // Perform vector addition
    if (tid < n)
        c[tid] = a[tid] + b[tid];
}
int main() {
    int n = 1024;  // Size of the vectors
    int *a, *b, *c; // Host vectors
    int *dev_a, *dev_b, *dev_c; // Device vectors

    // Allocate memory for the host vectors
    a = (int*)malloc(n * sizeof(int));
    b = (int*)malloc(n * sizeof(int));
    c = (int*)malloc(n * sizeof(int));
    // Allocate memory on the device
    cudaMalloc((void**)&dev_a, n * sizeof(int));
    cudaMalloc((void**)&dev_b, n * sizeof(int));
    cudaMalloc((void**)&dev_c, n * sizeof(int));
    // Initialize input vectors
    for (int i = 0; i < n; i++) {
        a[i] = i;
        b[i] = i;
    }
    // Copy input vectors from host to device
    cudaMemcpy(dev_a, a, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, n * sizeof(int), cudaMemcpyHostToDevice);

    // Define the block size and grid size
    int blockSize = 256;
    int gridSize = (int)ceil((float)n / blockSize);

    // Launch the kernel
    vectorAdd<<<gridSize, blockSize>>>(dev_a, dev_b, dev_c, n);

    // Copy the result vector from device to host
    cudaMemcpy(c, dev_c, n * sizeof(int), cudaMemcpyDeviceToHost);

    // Print the result
    for (int i = 0; i < n; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }
    // Cleanup
    free(a);free(b);free(c);
    cudaFree(dev_a);cudaFree(dev_b);cudaFree(dev_c);
    return 0;
}//COMPILE USING nvcc COMPILER
```

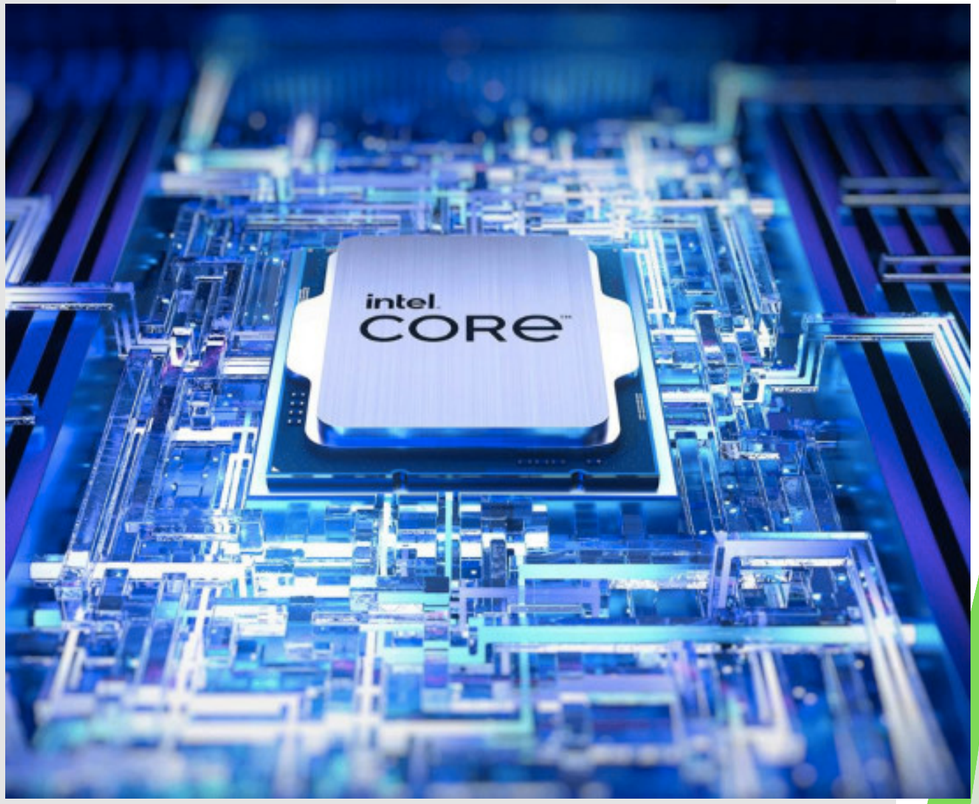
CPU Architecture

Cores:
CPUs have fewer powerful cores for efficient sequential processing and handling complex instructions.

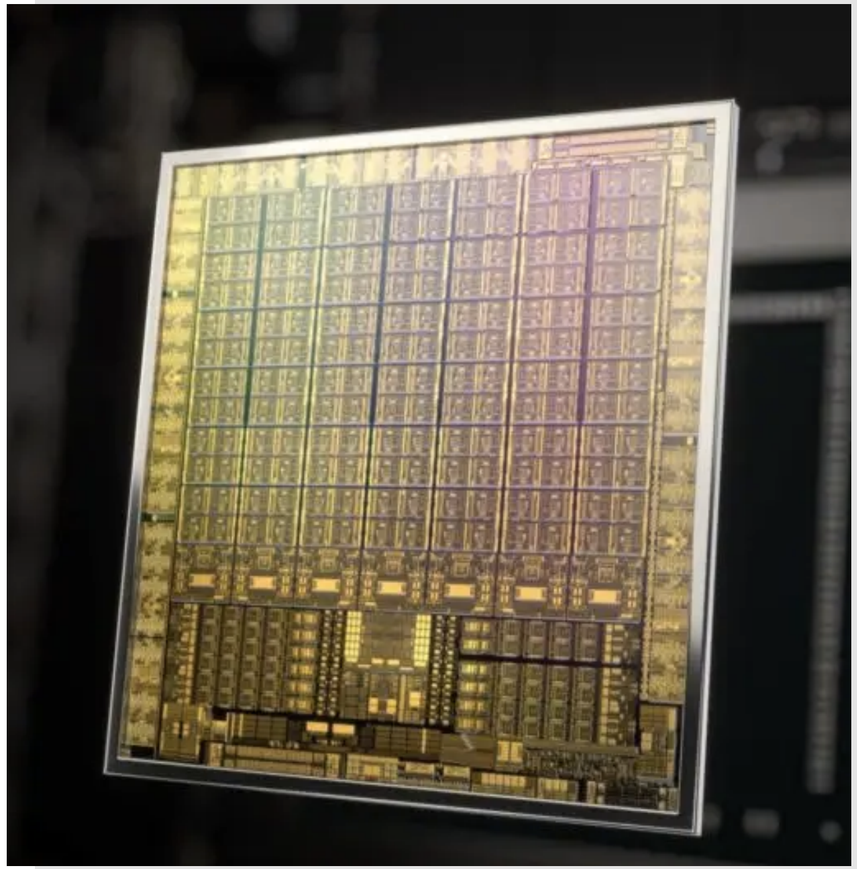
Memory Hierarchy:
CPUs have cache levels, a memory controller, and support for different memory types for efficient data access and management.

Clock Speed:
CPUs have high clock speeds for fast instruction execution, crucial for single-threaded and instruction-dependent tasks.

Cache:
CPUs have large caches to store frequently accessed data, instructions, and results, minimizing memory latency and boosting data access speed.



GPU Architecture



Memory Hierarchy:
GPUs have on-chip shared memory, registers for fast data access per core, and memory (VRAM) for data storage.

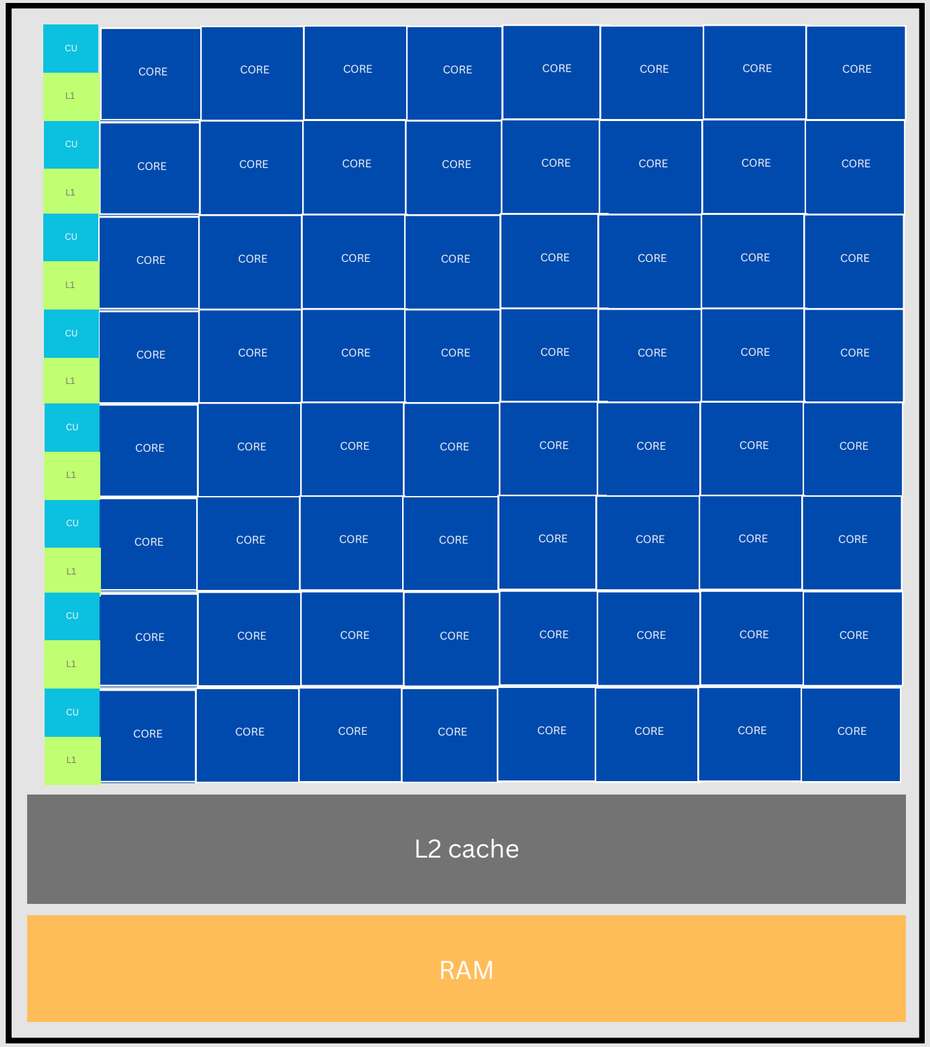
Cores:
GPUs have numerous smaller, power-efficient cores, which collectively deliver high computational power.

Memory Bandwidth:
GPUs prioritize memory bandwidth with wider buses and higher clock speeds, ensuring high throughput and reducing data transfer bottlenecks.

SIMD(Single Instruction, Multiple Data):
GPUs employ SIMD architecture for parallel processing of large datasets, executing instructions simultaneously on different data.

CPU ARCHITECTURE

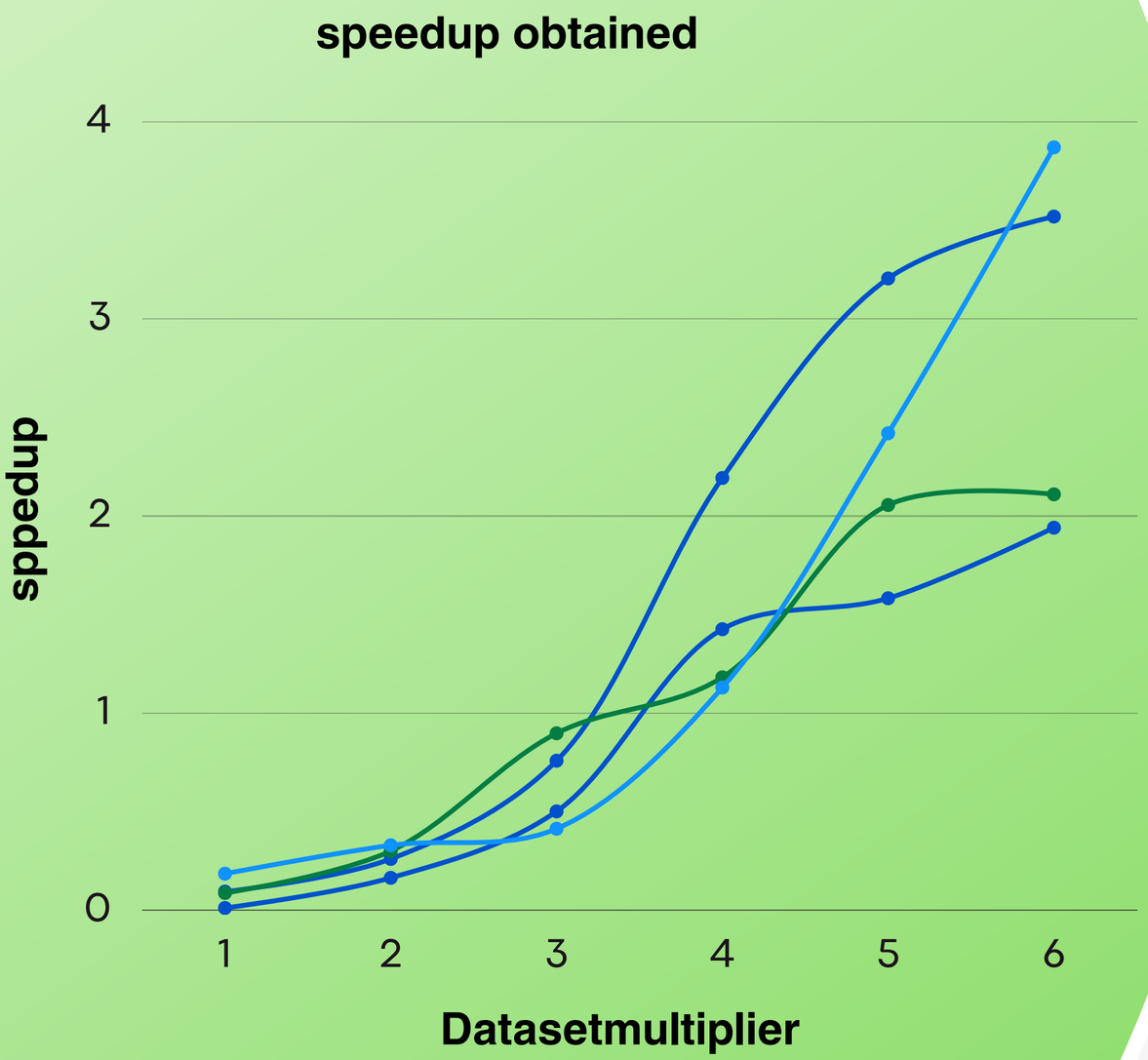
GPU ARCHITECTURE



ALGORITHMS EVALUATED

Algorithm	Type	Uses
Fuzzy Clustering	classification	Image segmentation, Pattern recognization, data classification etc
RSA Encryption	encryption	Secure Communication and Secure Key Exchange.
Zerotree Wavelet	compression	efficient lossy data compression
Bitonic Sort	sorting	Data Analysis, easier searching and easier data management.
Finite Difference Method	ODE solver	heat transfer, diffusion, field analysis, fluid dynamics problems etc

RESULT ANALYSIS



GPUs struggle when dataset size is too low

As dataset size increases GPUs perform better

Speedup can be increased with optimizations

Speedup is dependent on algorithms

ALGORITHMS WHICH SUIT GPU

Parallel computations:

GPUs have thousands of processing cores that can execute instructions concurrently, enabling simultaneous execution of independent tasks on different data elements. With their efficient memory access, GPUs can quickly read and write data.

Large-scale Data Processing:

such as big data analytics, scientific simulations, and numerical computations, often require high memory bandwidth. GPUs can efficiently handle these problems by parallelizing the computations across their many cores and accessing data from memory in parallel.

Floating-Point Performance:

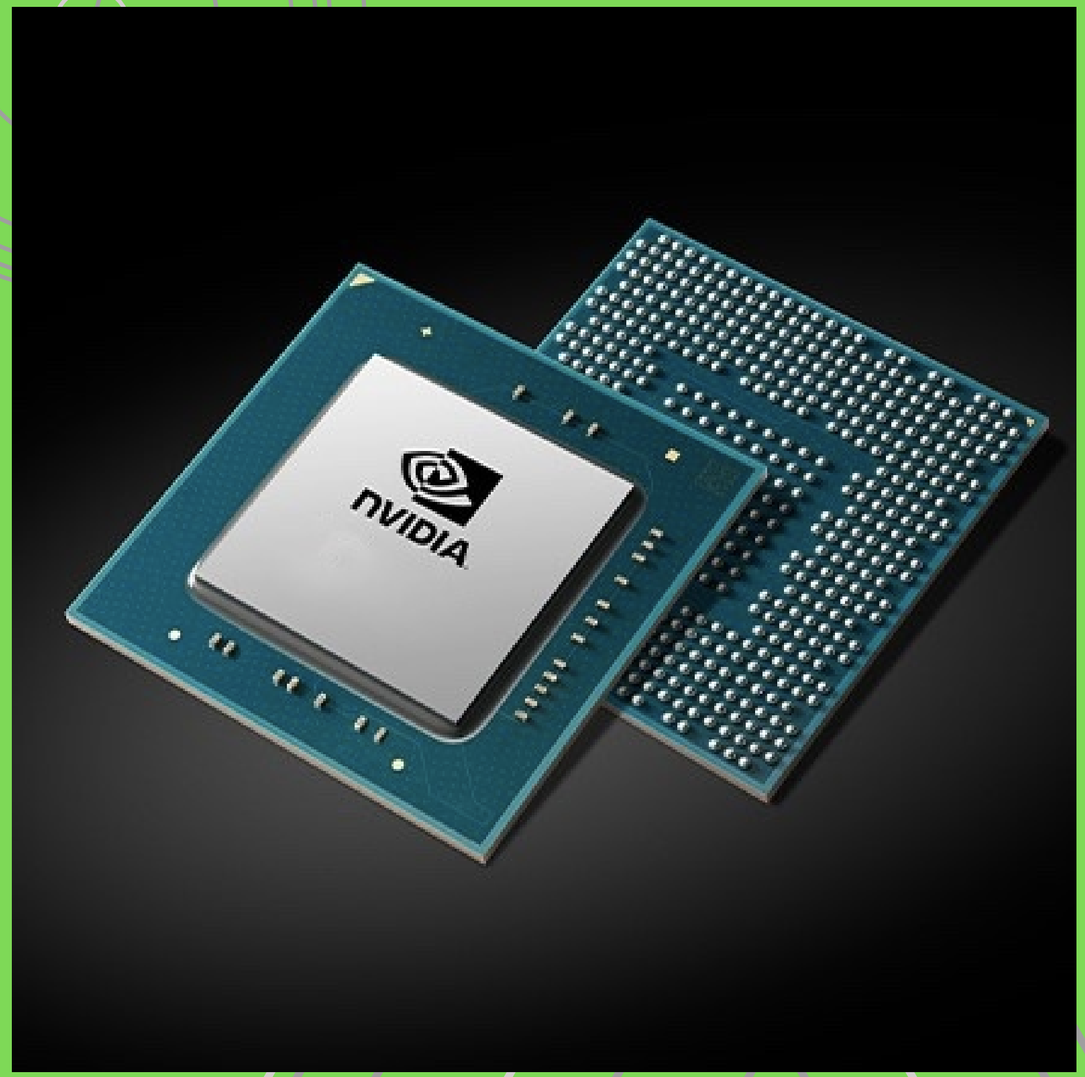
making them particularly well-suited for computationally intensive tasks that involve extensive mathematical calculations. They can perform a high number of floating-point operations per second.

ALGORITHMS AND CASES THAT ARE NOT FOR GPU

Sequential Algorithms:
Algorithms that have inherent sequential dependencies, where each step relies on the result of the previous step.

Branch-Heavy Algorithms:
Algorithms with a high degree of branching and conditional statements may not perform as well on GPUs.

Memory-Bound Algorithms:
Algorithms that heavily rely on memory access rather than computation or if the algorithm is limited by memory latency or has irregular memory access patterns.



ALGORITHMS AND CASES THAT ARE NOT FOR GPU

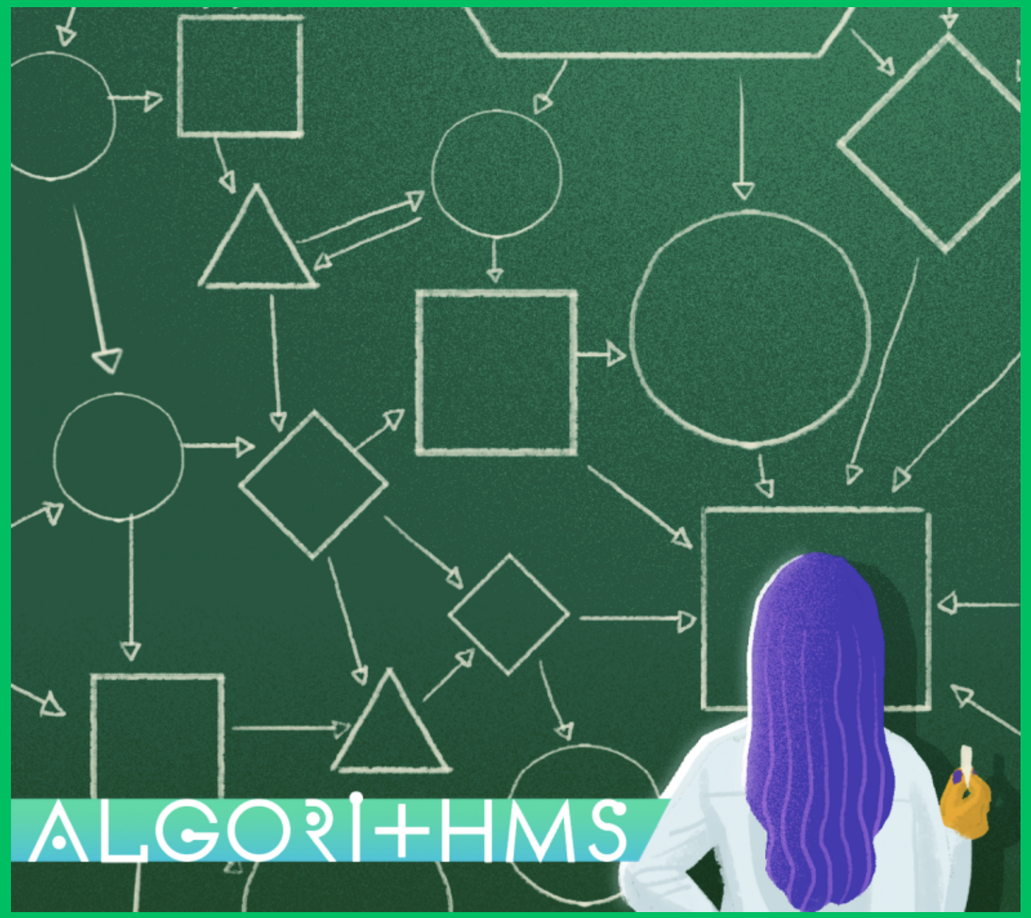
Small Data Sizes:
If the dataset or problem size is small, the overhead of transferring data to and from the GPU can outweigh the performance gains.

Latency-Sensitive Applications:
Real-time or latency-sensitive applications that require quick responses to user input or events may be better suited for CPUs,

Serial Dependencies:
where each iteration relies on the result of the previous iteration, may not be suitable for GPUs in all cases.

Algorithms with Irregular Workloads:
where the amount of work performed by each processing unit or thread varies significantly. GPUs are most efficient when workloads are well-balanced among threads.

Optimal algorithm design for GPU



Parallelism:
Identify opportunities for parallelism in your algorithm and structure your computations accordingly. Break down the problem into smaller independent tasks.

Memory access patterns:
Adjacent threads should access adjacent memory locations. Avoid irregular or scattered memory access patterns that can lead to memory bank conflicts.

Thread divergence:
Ensure that threads within a block follow the same execution path as much as possible to avoid performance penalties due to branching.

Memory hierarchy:
Exploit data reuse by storing frequently accessed data in shared memory or registers.

Optimal algorithm design for GPU

Optimizations:
Adapt your algorithm to leverage GPU-specific features. Consider GPU-specific libraries (e.g., cuBLAS for linear algebra)

Load balancing:
Combine multiple kernels into a single kernel if they operate on the same data to reduce kernel launch overhead and improve overall efficiency.

Asynchronous operations:
Exploit asynchronous execution by overlapping data transfers and computations. Utilize streams and events to overlap communication with the CPU and GPU computations.

Profiling and optimization:
Identify bottlenecks, analyze memory access patterns, and iteratively optimize your code based on the analysis results.

THANK YOU