Performance Comparison of Advanced Algorithms on CPU and GPU

Abstract

Parallel processing is a technique that involves dividing a large computational task into multiple smaller tasks that can be executed concurrently. Due to its architecture, which consists of several computation-capable cores, Graphics Processing Units (GPUs) are well-suited for parallel processing. The intricacy of the algorithm itself is key in evaluating whether a CPU or GPU is more appropriate. GPUs work best with algorithms that are highly parallelizable and allow for the division of calculations into separate jobs. However, algorithms with complicated control flow or sequential dependencies may perform better on CPUs due to their capacity to manage such complexities effectively. Algorithm complexity is just one element that affects how CPUs and GPUs compare in terms of performance.

Section 1: Introduction

The NVIDIA Jetson Nano Developer Kit is a powerful, portable computer made especially for creating and implementing cutting-edge artificial intelligence (AI) applications. It offers developers and hobbyists a platform to design and implement AI-powered projects across a range of industries, such as robotics, the Internet of Things, computer vision, and more.

The frequency and volume of data transfers between the CPU and GPU RAM should be taken into consideration since they might add overhead. Performance is also influenced by hardware specifications, such as the quantity of CPU cores or GPU compute units. In this study, we compared the performance of CPU and GPU while taking into consideration temporal variations and evaluating performance in GFlops.

Our primary evaluation is based on the NVIDIA Jetson Nano development kit and some additional analysis is done on the NVIDIA GeForce GTX 350 graphics card and Intel i5 quad-core CPU were utilized. This includes 10 benchmarks in favor of GPU from various fields, including data mining, sorting, and image compression algorithms. Each algorithm has been developed in CUDA, C++, and C to compare the amount of time spent on the CPU and GPU.

Section 2: Comparison of CPU and GPU

The designs of CPUs and GPUs are distinct, and they perform well at different kinds of workloads. While GPUs are made for parallel processing and excel at graphics-intensive jobs and computations that may be parallelized, CPUs are flexible and optimized for general-purpose computing and single-threaded workloads. Both processors are essential to contemporary computing, and their combined use may significantly boost performance across a range of tasks.

CPUs are designed with strong cores that are optimized for sequential processing, complicated instruction sets, and big caches. They are suitable for general-purpose computing and jobs that demand excellent single-threaded speed. On the other hand, GPUs feature a massively parallel design, smaller cores that are optimized for data parallelism, and high memory bandwidth. They excel in tasks that can be performed in parallel, such as rendering images, scientific simulations, machine learning, and artificial intelligence (AI), where they may carry out thousands of computations at once.

2.1 Architectural differences

CPUs are designed with a few powerful cores optimized for sequential processing. GPUs, on the other hand, feature hundreds or even thousands of smaller, more power-efficient cores optimized for parallel processing.

2.1.1 CPU Architecture:

- **Cores**: CPUs typically have fewer cores (ranging from 2 to 64 cores), but each core is more powerful and capable of handling complex instructions efficiently. These cores are designed for sequential processing and can execute a wide variety of tasks.
- **Cache**: CPUs have large caches (L1, L2, and sometimes L3) that store frequently accessed data, instructions, and intermediate results. This helps reduce memory latency and speeds up data access
- **Instruction Set:** CPUs have complex instruction sets that support a wide range of instructions and operations. They are optimized for general-purpose computing and handle tasks requiring complex control flow efficiently.
- **Clock Speed:** CPUs are designed for high clock speeds, enabling them to execute instructions quickly. This is particularly important for single-threaded tasks and tasks with dependencies between instructions.
- **Memory Hierarchy:** CPUs typically have a memory hierarchy with multiple levels of cache, a memory controller, and support for various memory types to efficiently manage data access and latency.

2.1.2 GPU Architecture:

- **Cores:** GPUs feature massively parallel architectures with hundreds or even thousands of smaller, more power-efficient cores. These cores are simpler than CPU cores but collectively deliver high computational power.
- **SIMD Execution**: GPUs execute instructions in a Single Instruction, Multiple Data (SIMD) fashion, where each core performs the same instruction on different data simultaneously. This architecture enables parallel processing of large datasets.
- **Memory Hierarchy:** GPUs have dedicated on-chip memory called shared memory and registers that facilitate fast data access for each core. They also have off-chip global memory (VRAM) and constant memory for higher-capacity data storage.
- **Memory Bandwidth:** GPUs prioritize memory bandwidth to feed their many cores with data. They use wider memory buses and higher memory clock speeds to achieve high throughput and minimize data transfer bottlenecks.
- **Memory Access Patterns:** GPUs are designed to handle data with regular memory access patterns, such as those found in graphics rendering or matrix operations.

2.2 Processing capabilities

GPUs outperform CPUs in terms of parallel processing capabilities, floating-point operations, and memory bandwidth. CPUs, on the other hand, provide strong single-threaded performance, support a wider range of specialized instructions, and are more power-efficient. The choice between CPU and GPU depends on the specific requirements of the task at hand and the trade-offs between single-threaded performance and parallel processing capabilities.

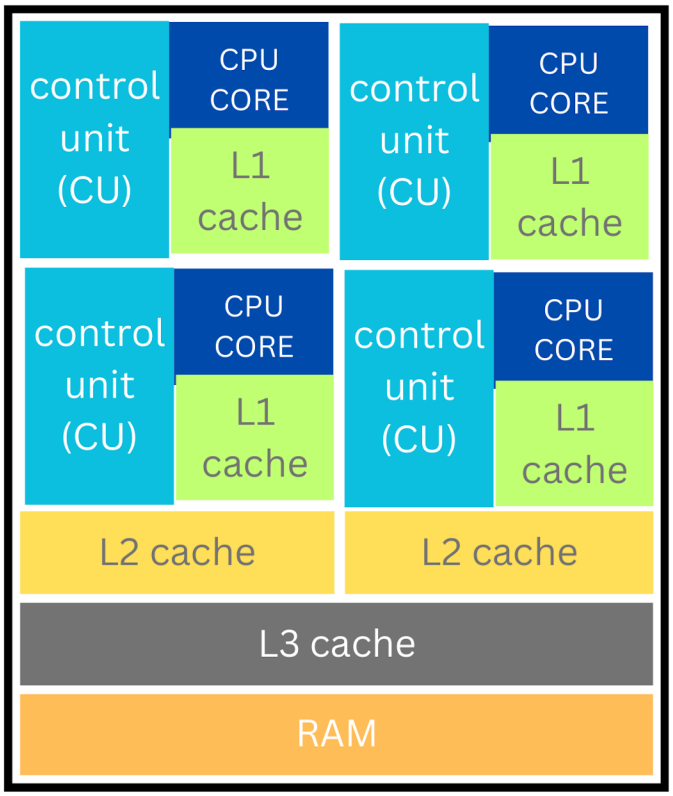


fig: 2.1 example of CPU architecture

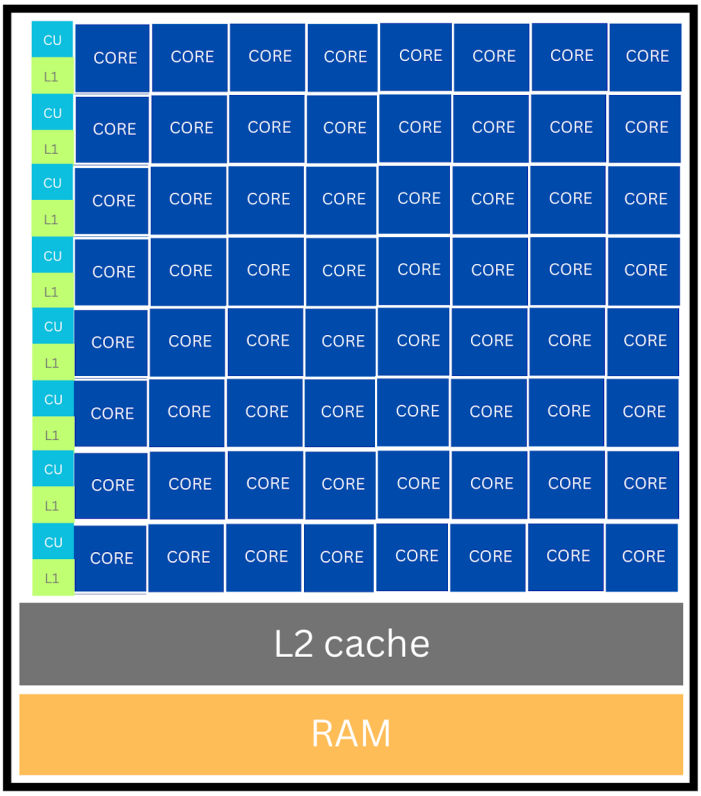


fig: 2.2 example of GPU architecture

2.2.1 Parallel Processing

GPUs are specifically designed for parallel processing. They feature hundreds or even thousands of smaller, more power-efficient cores that work in parallel to perform computations. CPUs, on the other hand, have fewer cores optimized for sequential processing, although modern CPUs also support some level of parallelism.

2.2.2 Floating-Point Operations

GPUs excel at performing floating-point operations, making them highly efficient for tasks involving complex calculations. They have dedicated hardware for floating-point arithmetic, enabling them to handle large-scale mathematical operations with high precision and speed. CPUs also support floating-point operations but typically have lower performance compared to GPUs in this regard.

2.2.3 Memory Bandwidth

GPUs generally offer significantly higher memory bandwidth compared to CPUs. This allows them to efficiently access and process large amounts of data simultaneously, which is particularly beneficial for tasks involving large datasets or memory-intensive operations.

2.2.4 Specialized Instructions

CPUs often provide a wide range of specialized instructions to handle various types of tasks efficiently. These instructions cover areas such as cryptography, multimedia processing, and specific algorithms. GPUs, while not as diverse in terms of specialized instructions, are highly optimized for graphics-related operations and parallel computations, such as those used in machine learning and scientific simulations.

2.2.5 Power Consumption

CPUs are designed to balance performance and power consumption, making them more power-efficient compared to GPUs. GPUs consume more power due to their higher number of cores and parallel processing capabilities. However, advancements in GPU architecture and power management techniques have made them more energy-efficient over time.

2.2.6 Task Optimization

CPUs are well-suited for general-purpose computing tasks and offer strong performance in a wide range of applications. They excel at tasks that require fast response times, single-threaded performance, and complex control flow.

Section 3: CUDA

3.1 CUDA programming

CUDA is a powerful parallel computing platform and programming model that allows developers to tap into the massive parallel processing power of NVIDIA GPUs. It provides a programming framework, memory management, and libraries that enable the acceleration of a wide range of computational tasks. CUDA has played a significant role in advancing GPU computing and has become a key technology for high-performance computing and GPU-accelerated applications. CUDA programming involves writing special functions called kernels. Kernels are executed in parallel by multiple threads on the GPU. Kernels are written in a C/C++-like language with additional syntax and directives specific to CUDA.

3.1.1 Parallel acceleration

CUDA, through the nvcc compiler, targets the GPU using the PTX (Parallel Thread Execution) Instruction Set Architecture. PTX instructions are generated by high-level language compilers.such as CUDA C/C++, CUDA Fortran, CUDA, and Python) and are optimized and translated to native target-specific GPU instructions.

The GPU code is organized as a sequence of kernels, which are functions executed in parallel on the GPU. Threads are grouped into blocks, and blocks are grouped into a grid. Each thread and block have a unique index, allowing access to specific array locations. The scheduling unit is a warp of 32 threads, and the maximum number of threads per block and simultaneous threads depends on the GPU architecture.

3.1.2 Basic CUDA Concepts

- **Thread:** A thread on the GPU is a basic element of the data to be processed. Unlike CPU threads, CUDA threads are extremely "lightweight," meaning that a context change between two threads is not a costly operation.
- **Block:** A Block is a set of threads that share a Stream Multiprocessor, its shared memory space, and thread synchronization primitives. Threads within a block coordinate by shared memory, atomic operations, and barrier synchronization.
- **Grid:** The grid is defined by specifying the number of thread blocks in each dimension, known as the grid dimensions. Each thread block within the grid can contain multiple threads.
- **Kernel:** A Kernel is a function that is meant to be executed in parallel on an attached GPU. Threads are utilized in CUDA to execute the kernel. Instead of being executed only once, a kernel is executed N times in parallel by N different threads on the GPU.
- **Wrap:** A warp in CUDA, is a group of 32 threads, which is the minimum size of the data processed in SIMD fashion by a CUDA multiprocessor.

3.2 CUDA interface calls

3.2.1 Memory Initialization

Linear Memory in CUDA is allocated using **cudaMalloc(),** which is similar to the malloc function in C.

Malloc returns the pointer to the block of memory that was dynamically allocated, and cudaMalloc takes the pointer of pointers and makes it point to the block of memory that was dynamically allocated. As we mentioned above, all memory allocated in the heap is stored in the global memory of the GPU.

The memory allocated is freed using **cudaFree()** and data transfer between host memory and device memory is typically done using **cudaMemcpy().** The last argument of cudaMemcpy determines the direction of the memory copy:

cudaMemcpyHostToDevice indicates copying from the device to the host.

cudaMemcpyDeviceToHost indicates copying from the device to the host.

3.2.2 Kernel Specifiers

- **__global__**: A kernel is defined with the **__global__** specifier and executed by multiple CUDA threads using the <<<...>>> syntax. Each thread has a unique thread ID accessible within the kernel through built-in variables.
- **__device__:** The **__device__** specifier declares a function that is executed on the device and can only be called from the device.

- **__shared__:** The **__shared__** specifier declares a variable that resides in the shared memory space of a thread block, has the lifetime of the block, has a distinct object per block, is only accessible from all the threads within the block, and does not have a constant address.
- **__constant__:** The **__constant__** specifier declares a variable that resides in constant memory space, has the lifetime of the CUDA context in which it is created, and has a distinct object per device.

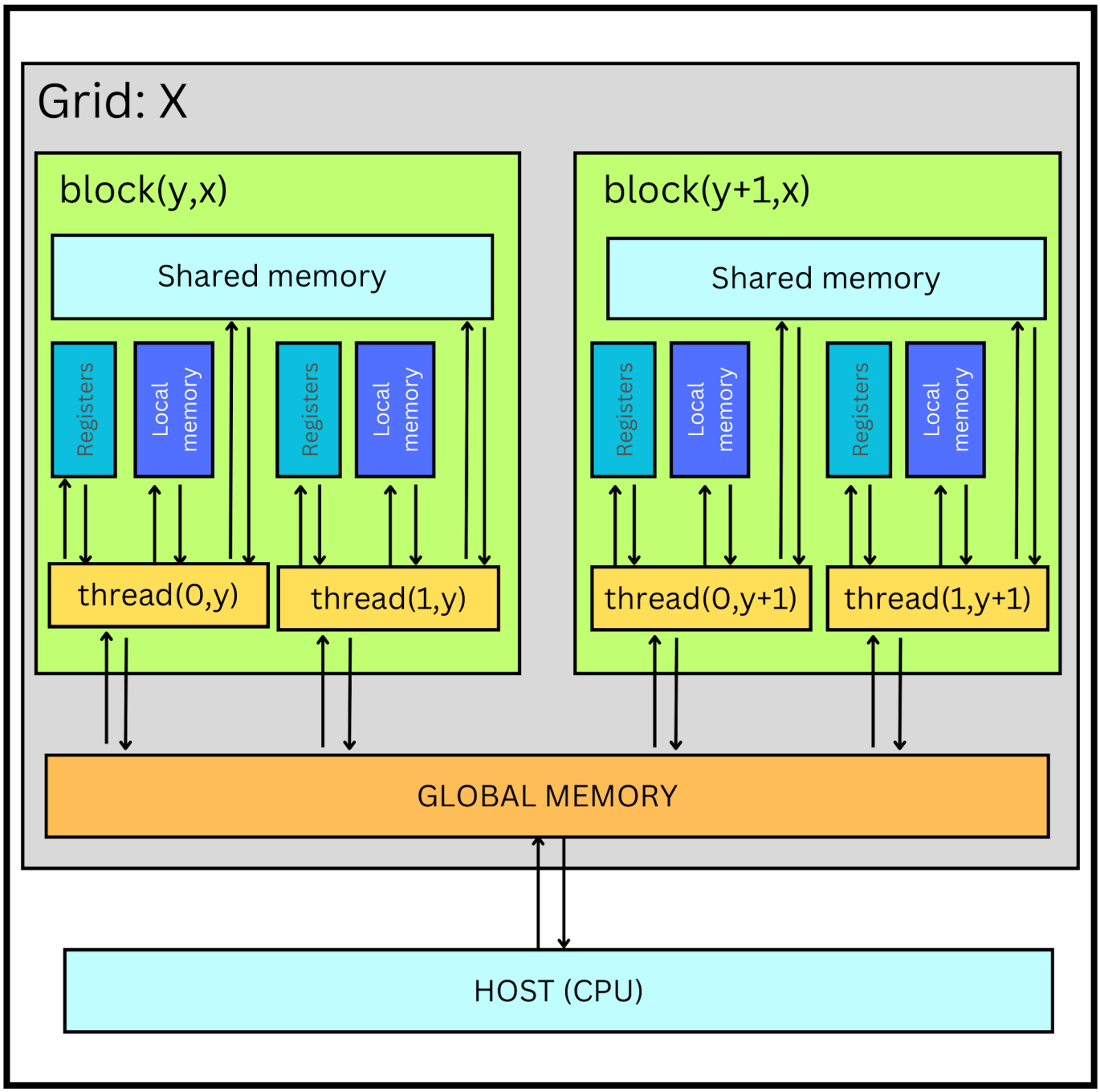


Figure 3.1 The above figure describes execution of threads in GPU. On chip shared and local memory are utilized by threads inside block, and blocks inside the grid synchronize to off-chip global memory. Global memory is shared between all GPU Kernels.

3.2.3 Built-In Specifiers

- **gridDim**: Contains Dimensions of the grid.
- **blockDim :** Contains Dimensions of the block.
- **blockIdx:** contains the block index within the grid (as shown in fig 3.1)
- **threadIdx:** contains the thread index within the block. (as shown in fig 3.1)
- **warp size:** contains the warp size in threads

3.2.4 Synchronization Tools

Block Level:

- **__syncthreads()**: The __syncthreads() function is used to synchronize threads within a thread block. It ensures that all threads in the block have reached that point and that memory accesses made prior to __syncthreads() are visible to all threads. It helps avoid data hazards when multiple threads access shared or global memory. It can be used in conditional code as long as the condition evaluates identically across the entire block.
- **__syncwarp()**: __syncwarp() synchronizes threads within a warp in CUDA. It ensures that all threads within the warp reach a certain point before any thread proceeds further, allowing for consistent memory visibility within the warp. Unlike __syncthreads(), which synchronizes all threads within a thread block, __syncwarp() is limited to synchronizing threads within a warp only.

Grid Level:

- **__threadfence():** threadfence() is a CUDA intrinsic function that enforces memory synchronization and ordering guarantees within a single thread. It ensures that memory operations issued by a thread before the threadfence() call are globally visible to all other threads in the GPU device after the threadfence() call. It prevents the reordering of memory operations across the threadfence() call and is commonly used to synchronize memory accesses between different threads or memory spaces.

3.3 CUDA performance considerations

- Minimize global memory accesses by utilizing shared memory and thread-level caching. Optimize memory access patterns to maximize memory coalescing, such as ensuring consecutive threads access consecutive memory locations
- Data Locality: Exploit data locality to reduce memory traffic. Reuse data as much as possible within the thread block by utilizing shared memory. Arrange data in memory to ensure co-located threads access adjacent memory locations.
- Thread Divergence: Minimize thread divergence within a warp (a group of threads executed together). Divergent execution paths within a warp can lead to performance penalties due to serialized execution. Try to ensure threads within a warp follow the same code path as much as possible.
- Overlapping Computation and Communication: Hide data transfer latencies by overlapping computation and communication. Use asynchronous memory copies (CUDA streams) to transfer data while performing computations concurrently.
- Kernel Fusion: Combine multiple CUDA kernels into a single kernel to reduce kernel launch overhead. Kernel fusion can help reduce memory access and improve performance by eliminating unnecessary data transfers between kernels.
- Thread and Block Organization: Choose the appropriate block and thread dimensions for your GPU architecture. Use a sufficient number of threads per block to fully utilize the GPU resources. Consider using 2D or 3D thread block organizations to improve memory access patterns.
- Loop Unrolling and Fusion: Unroll loops and fuse multiple loop iterations into a single iteration to reduce loop overhead. This technique can improve instruction-level parallelism and increase the number of active threads.
- Occupancy: Maximize the occupancy of the GPU by keeping a sufficient number of active threads per multiprocessor. Occupancy is crucial for utilizing all the available compute resources efficiently.

Section 4: Implementation and Setup

We used CUDA 10.2 to compile algorithms and sequential implementation is done in C language All algorithms are tested repetitively many times on devices then results were noted. We considered algorithms and programs which are used in many fields of engineering. Parallel implementation on GPU took a lot of optimizations which are not done on CPU. To display bad use cases of GPU we also used single-threaded and nonparallelizable algorithms But the results are not disclosed here. Our main goal is to extend the use of jetson boards or GPUs in all possible sectors. Our main work and research is done on Jetson Nano which comes under the category of embedded GPU which uses shared 4GB RAM with CPU. We also used a discrete GPU GeForce GTX 350 for further reference.

4.1 Devices used for testing

Device	Type	RAM	Cores
Arm cortex A-57	CPU	4GB	4
Tegra x1	GPU	4GB	128
intel i5	CPU	8GB	8
GeForce GTX 350	GPU	8GB	640

4.2 Algorithms used for evaluation

	Algorithm	Type	Description
1	Finite Difference Method	ODE solving	heat transfer, diffusion, field analysis, fluid dynamics problems etc
2	euler angles	mechanics	3d Body dynamics, graphics simulation etc
3	fuzzy clustering	classification	Image segmentation, Pattern recognization, data classification etc
4	RSA encryption	cryptology	Secure Communication and Secure Key Exchange.
5	zerotree wavelet	compression	efficient lossy data compression
6	BItonic Sort	sorting	Data Analysis, easier searching and easier data management.

Section 5: Evaluation

5.1 Finite Difference Method

FDMs are a class of numerical approaches for solving differential equations by approximating derivatives with finite differences To approximate the value of the solution at these discrete places, algebraic equations including finite differences and values from nearby points are solved. The spatial domain and time interval (if relevant) are both discretized, or broken into a finite number of steps.

(ODE) or (PDE), which may be nonlinear, are transformed by finite difference methods into a system of linear equations that can be resolved using matrix algebra. Due to the efficiency with which modern computers can carry out these linear algebra operations and the relative simplicity of their implementation, FDM is frequently used in contemporary numerical analysis.

5.1.1 Algorithm Implementation

Specify the heat conduction problem in rectangular coordinates, Determine the size of the grid (m x n) to discretize the domain.

Initialization:

- Set up the necessary parameters and variables for the problem,
- Create arrays to store the temperature values at each grid point (e.g., theta) and the guessed values (e.g., theta_guess).
- Initialize the temperature values.

Iterative Solution:

- Set up a loop to iteratively solve the heat conduction problem until convergence is reached.

Calculate Interior Points:

- Use nested loops to iterate over the interior points of the grid.
- Apply the finite difference equation to compute the temperature at each interior point based on neighboring values.
- Update the theta array with the new temperature values.

Calculate the Residual Error:

- Compute the difference between the current theta values and the corresponding theta_guess values.
- Accumulate the absolute differences to calculate the residual error sum.

Update Guess Values:

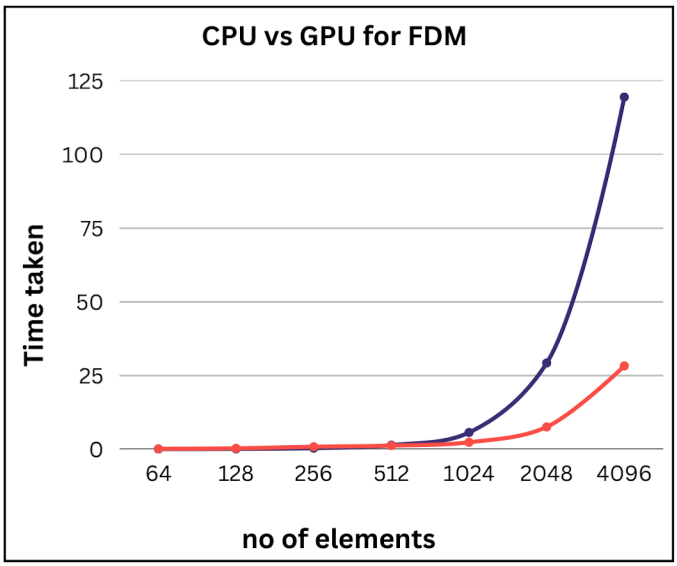
- Update the theta_guess array with the new guess values.

Check Convergence:

- Check if the residual error (sum) is below a specified tolerance or if the number of iterations has reached a maximum limit.
- If the convergence criterion is met, exit the loop

5.1.2 Results obtained

Matrix size	CPU	GPU	Speedup
64	0.026	0.140	0.186
128	0.094	0.285	0.33
256	0.345	0.832	0.414
512	1.369	1.203	1.13
1024	5.661	2.334	2.42
2048	29.223	7.537	3.87
4096	119.451	28.242	4.23



5.2 Computing Rotation matrix from Euler angles

The Euler angles are three angles that describe the orientation of a rigid body with respect to a fixed coordinate system
They can also represent the orientation of a mobile frame of reference in physics or the orientation of a general basis in 3-dimensional linear algebra. Euler angles can be defined by elemental geometry or by the composition of rotations. The geometrical definition demonstrates that three composed elemental rotations (rotations about the axes of a coordinate system) are always sufficient to reach any target frame.

5.2.1 Algorithm Implementation

Consider a, b, and c as our Euler's angle. Let Ra, Rb, and Rc be rotation matrices respectively for angles a, b, and c

$$\mathbf{Ra} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(a) & -sin(a) \\ 0 & sin(a) & cos(a) \end{pmatrix} \mathbf{Rb} = \begin{pmatrix} cos(b) & 0 & sin(b) \\ 0 & 1 & 0 \\ -sin(b) & 0 & cos(b) \end{pmatrix} \mathbf{Rc} = \begin{pmatrix} cos(c) & -sin(c) & 0 \\ sin(c) & cos(c) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Multiply the rotation matrices in the desired order to obtain the combined rotation matrix R:
R = Rz * Ry * Rx
The output matrix contains the input matrix after applying the rotation specified by the Euler angles (a, b, c).

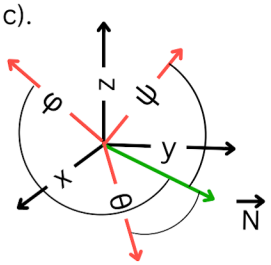


Figure 5.1 representation of euler angles of vector in free space

Sequential Implementation in C:

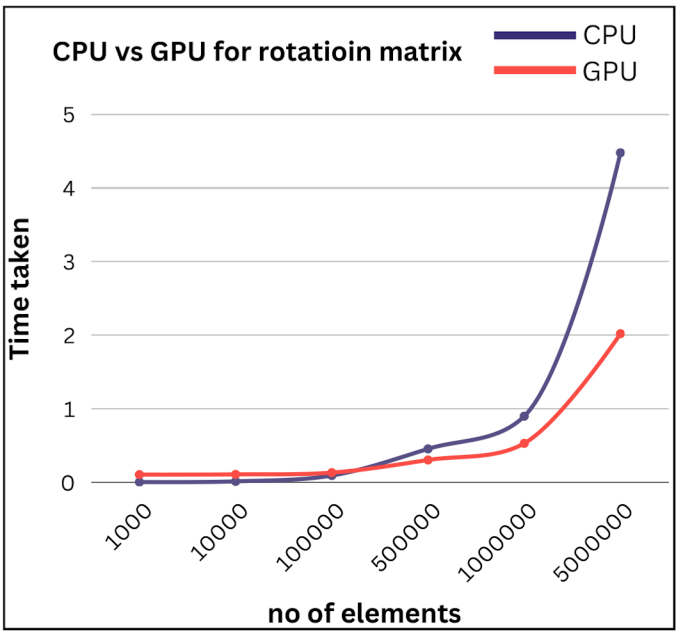
Initialize the Euler angles (roll, pitch, yaw) that describe the desired rotation.
Convert the Euler angles to radians if they are in degrees.
Calculate the sine and cosine values of the Euler angles using the math functions sin() and cos().
Construct the rotation matrix using the sine and cosine values. The elements of the rotation matrix are computed using the sine and cosine values of the Euler angles. The specific formula for constructing the rotation matrix depends on the order of rotation and coordinate system conventions. Use the rotation matrix to transform points or vectors by multiplying them with the rotation matrix.

Parallel Implementation in C:

Initialize the Euler angles (roll, pitch, yaw) that describe the desired rotation on the host (CPU) side.
Transfer the Euler angles from the host to the device (GPU) memory.
Allocate device memory for the rotation matrix.
Launch the kernel function on the device with appropriate configurations to perform parallel computations.
Perform the calculations for computing the rotation matrix within the kernel function using CUDA-specific syntax and math operations.
Transfer the rotation matrix from the device to the host memory.
Use the rotation matrix to transform points or vectors by multiplying them with the rotation matrix.
Free the allocated device memory.

5.2.2 Results obtained

Number of Elements	CPU	GPU	Speedup
1000	0.003	0.104	0.028
10000	0.012	0.107	0.112
100000	0.092	0.131	0.702
500000	0.454	0.303	1.498
1000000	0.898	0.529	1.697
5000000	4.478	2.019	2.217



5.3 Fuzzy clustering C-means

fuzzy c-means clustering, is a clustering algorithm that assigns data points to clusters based on their degree of membership rather than a hard assignment. Unlike traditional clustering algorithms that assign data points to a single cluster, fuzzy clustering allows data points to have partial memberships in multiple clusters. The c-means algorithm involves the following steps:

Update Membership Values: For each data point, the algorithm calculates the distance to each centroid and computes a ratio. The ratio reflects the degree of similarity between the data point and the centroid. The algorithm then raises the ratio to a power (controlled by the fuzziness parameter) and normalizes the sum of ratios to obtain the membership values.

Update Centroids: After updating the membership values, the algorithm recalculates the centroids based on the updated membership values. For each cluster, the algorithm calculates a weighted sum of the data points, where the weights are given by the membership values. The centroid is then normalized by dividing the sum by the total weight.

Convergence Check: The algorithm checks for convergence by calculating the maximum difference between the old and new membership values for each data point. If the maximum difference is below a certain threshold or the maximum number of iterations is reached, the algorithm terminates if not algorithm repeats again for convergence.

5.3.1 Algorithm Implementation

Sequential Implementation

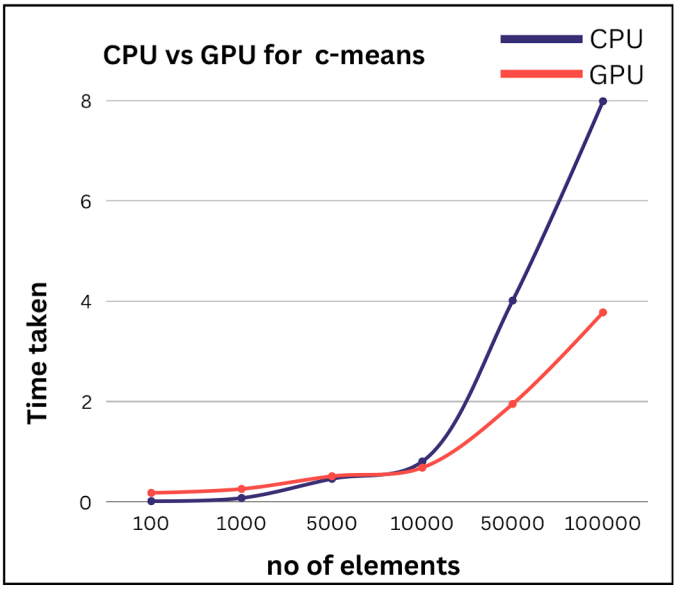
- Initialize the input dataset, the number of clusters, membership matrix, and the fuzziness parameter.
- Iterate until convergence: Update the cluster centers based on the current membership values. Update the membership matrix based on the current cluster centers and fuzziness parameters. Check for convergence criteria (e.g., maximum iterations or a small change in the membership values).
- Assign each data point to the cluster with the highest membership value.

Parallel Implementation in CUDA:

- Initialize the input dataset, the number of clusters, and the fuzziness parameter on the host (CPU) side. Transfer the dataset from the host to the device (GPU) memory.
- Allocate device memory for the membership matrix, cluster centers, and temporary variables.
- Launch the kernel function on the device with appropriate configurations to perform parallel computations. Each thread is responsible for updating the membership value for a specific data point and cluster. Multiple threads run in parallel, handling different data points and clusters simultaneously. Synchronization mechanisms may be required (e.g., atomic operations) to update shared variables.
- Check for convergence criteria within the kernel function and terminate if the conditions are met.
- Transfer the updated membership matrix and cluster centers from the device to the host memory.
- Assign each data point to the cluster with the highest membership value on the host side.

5.3.2 Results obtained

Number of Elements	CPU	GPU	Speedup
100	0.016	0.181	0.088
1000	0.078	0.259	0.301
5000	0.461	0.513	0.898
10000	0.806	0.683	1.182
50000	4.013	1.952	2.056
100000	7.988	3.778	2.110



5.4 Zerotree wavelet

The ZeroTree Wavelet (ZTW) algorithm is primarily used for compressing still images and is known for its effectiveness in achieving high compression ratios while maintaining good image quality. The Embedded zero wavelet algorithm achieves the compression by eliminating or quantizing less significant wavelet coefficients while retaining important information necessary for reconstruction. It can achieve large compression ratios by making use of the spatial redundancy and perceptual characteristics of images. The EZW algorithm utilizes a threshold to identify which wavelet coefficients are relevant during encoding and only explicitly encodes those significant coefficients. The idea of zerotrees, which enables bypassing the encoding of consecutive coefficients with very low magnitudes, is used to encode insignificant coefficients.
The encoded bitstream is utilized to recreate the wavelet coefficients and ultimately generate the compressed image during the decoding phase. The method itself is lossy since some data is lost through quantization and the elimination of less important coefficients.

5.4.1 Algorithm Implementation

Sequential implementation

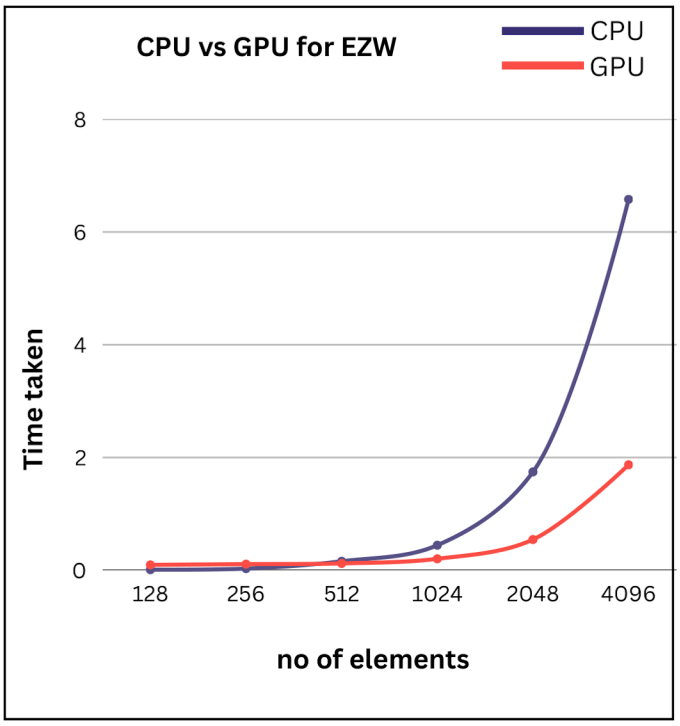
- Put the input image through the wavelet transform.
- Decide on a baseline threshold value.
- Repeat the wavelet coefficients in a particular sequence, such as decreasing magnitude.
- Each coefficient should be compared to the threshold:
- The coefficient is coded as significant (1) and the threshold is deducted from it if the coefficient magnitude is larger than or equal to the threshold.
- Encode the coefficient magnitude as insignificant (0) if it falls below the threshold.
- Based on the largest magnitude of the significant coefficients so far, adjust the cutoff value.
- From the encoded coefficients, produce a bitstream.

CUDA implementation

- Utilise CUDA kernels to apply the wavelet transform on the input image.
- Subband the image and save the wavelet coefficients in the device's memory.
- Perform zerotree encoding and thresholding as you go through the wavelet coefficients.
- To identify and encode relevant coefficients, use CUDA kernels.
- The threshold value should be updated, then the remaining coefficients should be encoded.
- The encoded coefficients should be used to create a compressed bitstream.
- Put the bitstream in the hardware's memory.
- Take the compressed bitstream out of the hardware memory.
- Reverse zerotree encoding and wavelet coefficient reconstruction are carried out using CUDA kernels.

5.4.2 Results obtained

matrix n*n	CPU(sec)	GPU(sec)	Speedup
128	0.009	0.094	0.095
256	0.028	0.107	0.261
512	0.158	0.120	0.759
1024	0.443	0.202	2.193
2048	1.744	0.544	3.205
4096	6.581	1.870	3.519



5.5 RSA encryption algorithm

It provides a secure method for encrypting and decrypting data by utilizing the properties of prime numbers and modular arithmetic. RSA is widely adopted for secure communication, digital signatures, and key exchange protocols. The RSA algorithm involves the following steps:

Key Generation:

Select two distinct prime numbers, typically large, denoted as p and q.
Compute the modulus n = p * q, which is the product of the two primes.
Calculate the totient of n as phi = (p - 1) * (q - 1).
Choose a public exponent e such that 1 < e < phi and gcd(e, phi) = 1, meaning e is coprime with phi.
Compute the private exponent d such that d ≡ e^(-1) (mod phi), i.e., d is the multiplicative inverse of e modulo phi.
The public key is (e, n) and the private key is (d, n).

Encryption:

Convert the plaintext message to a numerical representation, typically using a predetermined encoding scheme.
Given the public key (e, n), compute the ciphertext C as C ≡ M^e (mod n), where M is the plaintext message.

5.5.1 Algorithm Implementation

Sequential implementation

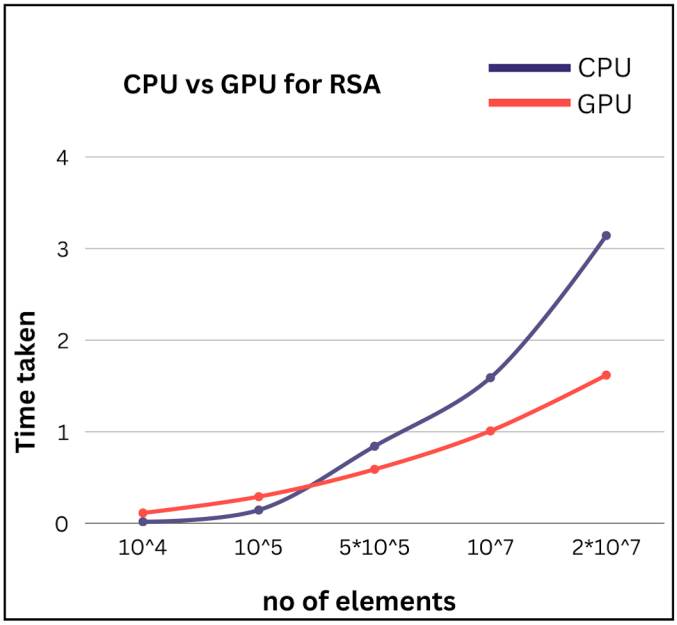
- Create function where the RSA encryption algorithm is implemented sequentially. The encryption step involves iterating over each element of the message array. For each element, the encryption formula C ≡ M^e (mod n) is applied, where M is the numerical representation of the message element, e is the public key, and n is the modulus. The result is stored in an array.
- Key Generation: The prime numbers p and q are selected by the user as input. The public key e and modulus n are then computed based on these prime numbers.
- Input Message: The user is prompted to enter the message, which is stored as a string. Each character of the message is then converted to its numerical representation using a predetermined encoding scheme. This numerical representation of the message is stored in an array called message.

CUDA implementation

- Create a kernel function to encrypt multiple message elements concurrently within a single thread. Each thread is responsible for encrypting a batch of elements, determined by the thread index and stride.
- The prime numbers p and q are selected by the user as input on the host (CPU) side.
- The public key e and modulus n are computed based on the selected prime numbers on the host side.
- The message is converted to a numerical representation on the host side.
- Memory allocation is performed on the device (GPU) side using cudaMalloc to allocate memory for the message and encrypted message arrays.
- The message array is transferred from the host to the device using cudaMemcpy.
- The encrypted message array is transferred back from the device to the host using cudaMemcpy.

5.5.2 Results obtained

message length	CPU	GPU	Speedup
10000	0.019	0.115	0.165
100000	0.147	0.293	0.502
500000	0.843	0.591	1.426
10000000	1.591	1.010	1.583
20000000	3.142	1.619	1.941



5.6 Bitonic Sort

Bitonic merge sort is a parallel sorting algorithm that is commonly used for sorting large numbers of elements on architectures with multiple parallel execution units, such as GPUs. It is also employed as a construction method for creating sorting networks. The algorithm produces sorting networks with O (n log^2(n)) comparators, where n represents the number of items being sorted. The delay of the resulting sorting networks is O (n log^2(n)), which indicates the number of steps needed for the sorting process. This efficiency and parallel nature make it an attractive choice for sorting tasks on architectures that support extensive parallelism like GPU's

5.6.1 Algorithm Implementation

Sequential Implementation:

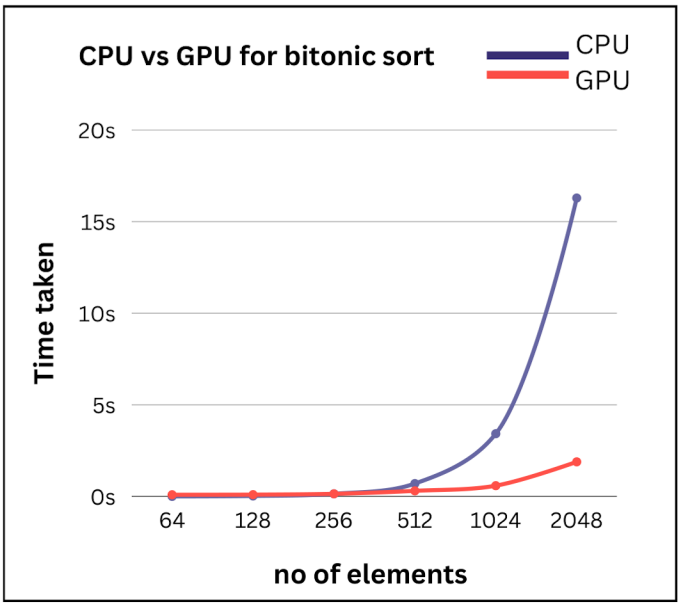
- **Divide and conquer:** Bitonic Sort is a recursive algorithm that divides the array into two halves, each representing a bitonic sequence.
- **Bitonic merge:** After dividing the array, the algorithm performs bitonic merges, which involve comparing and swapping elements between pairs of sub-sequence
- **Recursive calls:** The C implementation typically uses recursive calls to sort the two halves of the array and then performs bitonic merges to combine the sorted sub-sequences.

CUDA Implementation:

- **GPU parallelism:** The CUDA implementation takes advantage of the parallel architecture of GPUs by assigning each thread to handle a specific element in the array.
- **Kernel function:** A CUDA kernel function, such as bitonic Sort, is defined to perform the comparison and swapping operations within each thread.
- **Grid and block dimensions:** The grid and block dimensions are set to determine the number of parallel threads and how they are organized in the GPU.
- **Memory transfers:** The input array is copied from the host (CPU) to the device (GPU) memory using functions like cudaMalloc and cudaMemcpy. The sorted array is transferred back to the host memory after the sorting is complete.
- **Launching the kernel:** The kernel function is launched on the GPU, specifying the grid and block dimensions.
- **Synchronization:** cudaDeviceSynchronize() is used to ensure that all GPU threads have finished executing before proceeding to the next iteration or transferring data.
- **Iterative sorting:** The Bitonic Sort algorithm is implemented iteratively, with each iteration performing a different combination of stride and stage values to sort the array.

5.6.2 Results obtained

Matrix size (n*n) n=	CPU	GPU	Speedup Achieved
64	0.013s	0.103s	0.126
128	0.037s	0.108s	0.343
256	0.154s	0.153s	1.006
512	0.713s	0.316s	2.25
1024	3.434s	0.598s	5.75
2048	16.291s	1.898s	8.58



Section 6: Conclusions and Discussions

6.1 Algorithms which suit GPU

We already know that Matrix processing is much more efficient on GPUs than on CPUs because we can parallelize the execution of each matrix member. However, from some of the results, we conclude that there is a significant trade-off between memory transfer and execution time when the dataset is small. Anyone who is new to CUDA parallel programming may have attempted **simple vector addition**. We find there regardless of dataset size, sequential execution in CPU and parallel execution in GPU take the same amount of time or even less. Even though GPUs compute the answers in this scenario instantly, there is a cost associated with moving data from the host to the device and vice versa. This indicates that the process should at least be somewhat compute-bound even while parallel execution is possible. Sequential tasks, with heavy control flow, or those with limited parallelism may not see significant benefits from using GPUs over CPUs. The suitability of GPU acceleration depends on the specific workload, algorithm, and the ability to effectively parallelize computations.
In conclusion, any parallel non-divergent algorithms work well on GPUs. They can work even better with this capabilities

- **Parallel Processing Power:** GPUs feature a massively parallel architecture with hundreds or thousands of smaller, more power-efficient cores. This makes them highly effective in handling tasks that can be parallelized, such as graphics rendering, machine learning, and scientific simulations.
- **High Memory Bandwidth:** GPUs typically offer significantly higher memory bandwidth compared to CPUs. This allows for efficient data transfer and processing of large datasets, contributing to their high computational throughput.

6.2 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, may not be well-suited for GPUs. GPUs are designed for parallel processing, and they may not provide significant speedup for algorithms that cannot be effectively parallelized.

Branch-Heavy Algorithms: Algorithms with a high degree of branching and conditional statements may not perform as well on GPUs. GPUs are optimized for executing the same instruction across multiple data elements simultaneously. Frequent divergences in execution paths within a GPU thread block can hinder performance.

Memory-Bound Algorithms: Algorithms that heavily rely on memory access rather than computation may not benefit significantly from GPU acceleration. GPUs have high memory bandwidth, but if the algorithm is limited by memory latency or has irregular memory access patterns, the overall performance improvement may be limited.

Small Data Sizes: GPUs are designed to handle large amounts of data and parallel computations. If the dataset or problem size is small, the overhead of transferring data to and from the GPU can outweigh the performance gains. In such cases, the overhead associated with GPU memory management and data transfer can make CPUs more efficient.

Latency-Sensitive Applications: GPUs prioritize throughput and parallel processing over low-latency responses. Real-time or latency-sensitive applications that require quick responses to user input or events may be better suited for CPUs, which have lower latency and can handle single-threaded tasks more efficiently.

Serial Dependencies: Algorithms that have strong dependencies between iterations, where each iteration relies on the result of the previous iteration, may not be suitable for GPUs. These algorithms cannot be easily parallelized, and the GPU's parallel processing power may not be effectively utilized.

Algorithms with Irregular Workloads: Some algorithms have inherently 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. Irregular workloads can result in underutilization of GPU resources

6.3 Optimal algorithm design for GPU

The design might vary from GPU to GPU but it is important to acknowledge these conclusions these

Parallelism: GPUs excel at parallel processing. Identify opportunities for parallelism in your algorithm and structure your computations accordingly. Break down the problem into smaller independent tasks that can be executed simultaneously on different GPU cores.

Memory access patterns: Optimize memory access patterns to maximize memory throughput. Access memory in a coalesced manner, where adjacent threads access adjacent memory locations. Avoid irregular or scattered memory access patterns that can lead to memory bank conflicts.

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

Memory hierarchy: GPUs have different memory levels with varying access latencies and bandwidths. Utilize shared memory and local memory effectively to reduce global memory accesses. Exploit data reuse by storing frequently accessed data in shared memory or registers.

Optimizations: Adapt your algorithm to leverage GPU-specific features. For example, use vectorized operations, exploit data parallelism, and consider GPU-specific libraries (e.g., cuBLAS for linear algebra) to offload computations to highly optimized GPU implementations.

Load balancing: Strive for balanced workloads among GPU cores to maximize GPU utilization. Irregular workloads can result in some cores idling while others are still processing, leading to underutilization.

Kernel fusion: 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: Profile your GPU code using performance analysis tools provided by GPU vendors (e.g., NVIDIA's CUDA Profiler). Identify bottlenecks, analyze memory access patterns, and iteratively optimize your code based on the analysis results.