GPU Programming

Mihai Lucian Rîtan

June 2024

Contents

1 Introduction

The main goal of this document is to better understand GPU programming. Initially, essential aspects of CPU/GPU architecture will be presented, followed by examples and the thought process involved.

This document emphasizes the CISC x86/x64 architecture on the CPU side, as this is the architecture of the machine on which the presented examples were run. It is also one of the most popular architectures for personal computers. On the GPU side, the emphasis is on SIMD architecture, focusing on APIs that utilize CUDA cores from NVIDIA video cards.

At this moment, the document does not present examples with more general APIs such as OpenCL or OpenACC. It does not focus on or present graphical aspects such as shaders, ray casting, rendering APIs like Vulkan, OpenGL, or DirectX, nor any other logic related to graphical processing. Instead, the focus is on the highly parallelized nature of the multi-core architecture of GPUs and how we can use the GPU instead of the CPU for potentially better computing times for algorithms.

All experiments were done on a machine with the following specifications:

- CPU: i7-13700F:
  - Frequency: 2.1 Ghz (Base)/5.2 (Turbo)
  - Cores: 16 (8 performance core + 8 efficient cores)
  - L1 cache: 1.4 MB
  - L2 cache: 24.0 MB
  - L3 cache: 30.0 MB
- GPU: Nivida GeForce RTX 4070TI
  - Memory: 12 GB - GDDR6X - 21008 MHz
  - CUDA cores: 7680
- RAM: 32 GB DDR5 - 4800 MHz - Dual chanel (16+16)

Performance core means each thread has 2 logical processors and efficient means it has only one. So the system has a total of 24 logical processors.

2 Architectures

2.1 CPU Architecture

CPUs are designed to excel at executing a sequence of operations, called a thread, as fast as possible. Modern CPUs can execute a few tens of these threads in parallel. As of 2024, it is usual to see CPUs with clock speeds around 3-4 GHz per core. The number of cores can range from 2 to 96, but the most common range is from 2 to 12 cores.

2.1.1 Cache Memory

One of the critical components of a CPU's performance is its cache memory. Cache memory is a small, high-speed storage located close to the CPU cores, designed to store frequently accessed data and instructions to reduce the time it takes to fetch them from the main memory (RAM). There are typically three levels of cache in modern CPUs:

- **L1 Cache:** This is the smallest and fastest cache, located closest to the CPU cores. Each core usually has its own dedicated L1 cache, divided into two parts: one for instructions (L1i) and one for data (L1d). The size of L1 cache typically ranges from 32KB to 128KB per core.
- **L2 Cache:** This cache is larger and slightly slower than the L1 cache. Each core may have its own L2 cache, or it may be shared among a few cores. The size of L2 cache ranges from 256KB to 1MB per core.
- **L3 Cache:** This is the largest and slowest cache, but still much faster than main memory. L3 cache is usually shared among all the cores on a CPU. Its size can range from 2MB to 64MB or more, depending on the CPU design.

2.1.2 Cache Control

The CPU uses sophisticated algorithms to manage the cache, deciding which data to store and which to evict when the cache is full. The control mechanisms include:

- **Cache Coherence:** Ensures that changes in data values in one cache are propagated throughout the system, so all cores have the most up-to-date data.
- **Cache Replacement Policies:** Determine which cache lines to replace when new data is loaded into the cache. Common policies include Least Recently Used (LRU) and First In, First Out (FIFO).
- **Prefetching:** Predicts the data and instructions that will be needed next and loads them into the cache in advance to reduce latency.

In the Fig. 1 we can see an abstraction of a CPU architecture with 3 cache levels and 4 cores.

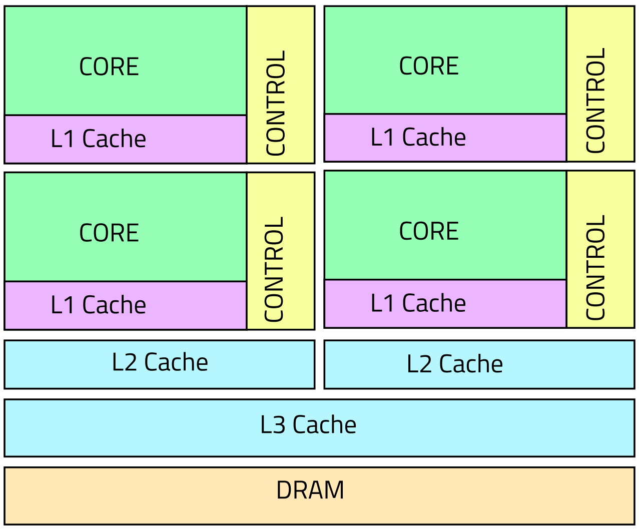

Figure 1: CPU architecture

2.1.3 CPU Architecture Examples

There are many architectures for CPUs that employ different approaches in CPU design philosophy to achieve various end goals, such as reducing power consumption or enhancing performance. Below are some notable examples:

- **CISC (Complex Instruction Set Computer):**
  - **Design Philosophy:** Aims to complete tasks in as few lines of assembly as possible. The instruction set is complex, allowing for multi-step operations within a single instruction.
  - **Example:** x86 architecture used in most desktop and laptop computers.
  - **Characteristics:** Complex instructions, variable-length instruction formats, and a large number of addressing modes.
- **RISC (Reduced Instruction Set Computer):**
  - **Design Philosophy:** Simplifies the instructions such that each instruction performs only a single task. This leads to simpler, more predictable instruction execution.
  - **Example:** MIPS, SPARC.
  - **Characteristics:** Simple instructions, fixed-length instruction formats, and fewer addressing modes.
- **ARM (Advanced RISC Machines):**
  - **Design Philosophy:** Similar to RISC with enhancements to support low power consumption, making it suitable for mobile and embedded systems.
  - **Example:** ARM Cortex series used in smartphones, tablets, and many embedded systems.
  - **Characteristics:** Efficient instruction execution, low power consumption, and a large register set.

2.2 GPU Architecture

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. A standard GPU typically has anywhere from a few hundred to several thousand cores.

Devoting more transistors to data processing, such as floating-point computations, is beneficial for highly parallel computations. The GPU can hide memory access latencies with computation instead of relying on large data caches and complex flow control to avoid long memory access latencies, both of which are expensive in terms of transistors.

2.2.1 Cache Memory

Unlike CPUs, GPUs have a different cache architecture designed to handle the massive parallelism. Here are the typical cache levels in a GPU:

- **L1 Cache:** Each multiprocessor in a GPU has its own L1 cache, which is smaller and more focused on handling specific threads within a block.
- **L2 Cache:** This cache is larger and shared among all the multiprocessors in the GPU. It helps in coordinating data that is used across different cores.

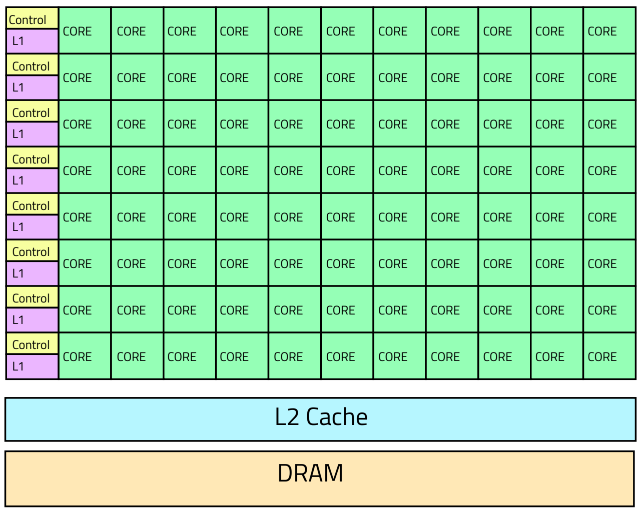

Figure 2: GPU architecture

3 CUDA Programming Model

The CUDA programming model is designed to leverage the highly parallel nature of GPU hardware. The key components of this model include Kernels, Threads, and Blocks, which allow for the efficient execution of parallel tasks.

3.1 Kernels

A kernel is a function written in CUDA C/C++ that runs on the GPU. When a kernel is launched, it is executed N times in parallel by N different GPU threads, as opposed to only once like a regular C/C++ function on the CPU. The kernel function specifies the code to be executed by each thread.

3.2 Threads

Threads are the basic unit of execution in the CUDA programming model. Each thread executes the kernel function independently. Threads are lightweight and designed to have minimal overhead, allowing thousands of threads to run concurrently on the GPU. Each thread has a unique thread ID that it can use to compute memory addresses and make control decisions.

3.3 Blocks

Threads are grouped into blocks. Each block contains a number of threads that can cooperate by sharing data through shared memory and synchronizing their execution to coordinate memory accesses. Blocks are executed independently, allowing CUDA to efficiently manage and schedule their execution on the GPU hardware.

3.4 Grid

Blocks are organized into a grid. A grid is a collection of blocks that can be one-dimensional, two-dimensional, or three-dimensional, depending on the application requirements. The grid allows for organizing large computations by defining a structured hierarchy of threads and blocks.

3.5 Memory Hierarchy

The CUDA programming model also defines a memory hierarchy that is crucial for optimizing performance (see fig. 3):

- **Global Memory:** Accessible by all threads, but has high access latency.
- **Shared Memory:** Shared among threads within the same block, allowing for fast data exchange.
- **Local Memory:** Private to each thread.
- **Constant and Texture Memory:** Read-only memory spaces that are cached for optimal performance.

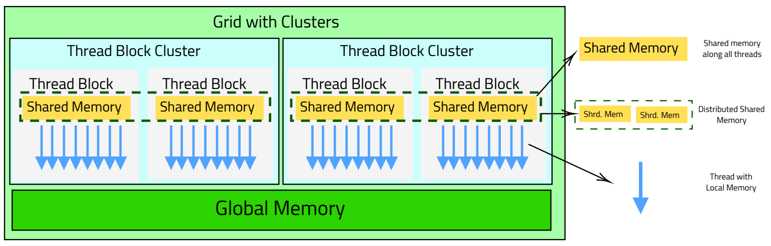

Figure 3: Memory Hierarchy

4 Thought Process

When considering the parallelization of an algorithm or code for execution on CPUs, GPUs, and particularly for a CUDA implementation, several factors and thought processes need to be considered to ensure efficient and effective utilization of the hardware resources.

4.1 Parallelization Considerations for CPUs

- **Task Decomposition:** Break down the algorithm into smaller tasks that can be executed in parallel. Identify independent tasks that can run concurrently.
- **Data Dependencies:** Analyze the dependencies between different tasks to avoid race conditions and ensure correct execution order.
- **Synchronization:** Use synchronization mechanisms such as mutexes, semaphores, or atomic operations to manage access to shared resources and prevent data corruption.
- **Load Balancing:** Distribute the tasks evenly across the available CPU cores to maximize utilization and minimize idle time.
- **Cache Utilization:** Optimize the algorithm to make efficient use of the CPU cache hierarchy (L1, L2, L3) to reduce memory access latency.

4.2 Parallelization Considerations for GPUs

- **Massive Parallelism:** Exploit the thousands of cores available on a GPU by designing algorithms that can be broken down into many parallel tasks.
- **Memory Access Patterns:** Optimize memory access patterns to maximize the use of high-bandwidth memory and minimize latency. Use coalesced memory accesses when possible.
- **Thread Divergence:** Avoid thread divergence within a warp (group of threads) by ensuring that threads in a warp follow the same execution path as much as possible.
- **Shared Memory Usage:** Utilize shared memory within a block for frequently accessed data to reduce the number of accesses to global memory.
- **Occupancy:** Maximize the number of active warps on a multiprocessor to fully utilize the GPU's resources and hide memory latency.

4.3 Parallelization Considerations for CUDA

When implementing an algorithm using CUDA, the following specific considerations should be taken into account:

- **Kernel Design:** Design kernel functions that can be executed by many threads in parallel. Each thread should perform a small part of the overall computation.
- **Thread Hierarchy:** Organize threads into blocks and grids. Choose an appropriate block size and grid size to optimize performance based on the problem size and hardware constraints.
- **Memory Hierarchy:** Utilize the CUDA memory hierarchy efficiently:
  - **Global Memory:** Used for data that needs to be accessed by all threads. Minimize accesses due to high latency.

    - **Shared Memory:** Used for data shared within a block. It has much lower latency than global memory.
    - **Local Memory:** Used for data private to each thread.
    - **Constant and Texture Memory:** Read-only memories with caching mechanisms to improve performance.

- **Thread Synchronization:** Use synchronization primitives such as `__syncthreads()` to coordinate threads within a block.
- **Performance Profiling:** Profile the CUDA code using tools like NVIDIA Visual Profiler or Nsight to identify performance bottlenecks and optimize accordingly.
- **Memory Transfers:** Minimize the overhead of memory transfers between the host (CPU) and the device (GPU). Use asynchronous memory transfers to overlap computation with data transfer.

By considering these factors, developers can effectively parallelize their algorithms to leverage the full potential of both CPU and GPU architectures, particularly when using CUDA for GPU programming.

5 Programming

In this section, we will take a more hands-on approach with code examples, comparisons, and experiments using multiple libraries and different kinds of problems and algorithms. The examples are exclusively done in the Python programming language with the following libraries for paralelization: CuPy, Numba

5.1 Addition of Two Vectors

The problem statement is simple: we have two 1-dimensional structures (vectors) with random real numbers, and we want to obtain a new structure by adding the corresponding values from the initial structures.

Given two vectors **A** $= [a_1, a_2, \ldots, a_n]$ and **B** $= [b_1, b_2, \ldots, b_n]$, the resulting vector **C** is obtained by:

$$\mathbf{C} = \mathbf{A} + \mathbf{B}$$

where each element c_i in **C** is computed as:

$$c_i = a_i + b_i \quad \text{for } i = 1, 2, \ldots, n$$

Example; Consider the following vectors:

$$\mathbf{A} = [1.2, 3.4, 5.6]$$

$$\mathbf{B} = [0.8, 2.2, 4.4]$$

The resulting vector **C** will be:

$$\mathbf{C} = [2.0, 5.6, 10.0]$$

5.1.1 C++ CUDA implementation

In Fig. 4 we can see the solution written in C++. Starting from this snippet, more Python examples will be provided.

What differentiates this code from a single-threaded alternative is the use of `threadIdx.x` and the way we call the method `VecAdd<<<1, N>>>(A, B, C)`. Let's discuss both of them.

Details regarding the thread:

- **Thread Index Calculation**: Each thread within a block has a unique index ranging from 0 to the number of threads per block minus one. This index is given by threadIdx.x.
- **Element-wise Operation**: The index i is used to access the elements of the vectors A, B, and C. Each thread adds the elements at position i of A and B and stores the result in C at the same position.
- **Parallel Execution**: Since the kernel is launched with multiple threads, each thread performs this operation in parallel, allowing the entire vector addition to be completed much faster than a sequential approach.

The syntax `VecAdd<<<1, N>>>(A, B, C)` is used to launch the kernel on the GPU. This syntax specifies the configuration of the kernel execution:

- **Grid Dimensions:** The first parameter inside the triple angle brackets (`<<<...>>>`) specifies the number of blocks in the grid. In this case, 1 indicates that there is a single block.

- **Block Dimensions:** The second parameter specifies the number of threads per block. In this case, N indicates that the block contains N threads.
- **Kernel Arguments:** The parameters inside the parentheses ((...)) are the arguments to the kernel function. Here, A, B, and C are pointers to the input and output vectors in device memory.

By launching the kernel with N threads, each thread can process a different element of the vectors in parallel, resulting in a significant performance improvement over a single-threaded approach.

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

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figure 4: Addition of two vectors with CUDA in C++
Source: Nvidia documentation https://docs.nvidia.com/cuda

5.1.2 Python CPU Sequential/Parallel implementation

Firstly let's see the implementation of addition of two vectors sequentially.

```python
import numpy as np

def vector_add_single_thread(a, b, c):
    for i in range(a.size):
        c[i] = a[i] + b[i]

# n values ex: 10 ** 5, 10 ** 6, 10 ** 7, 10 ** 8
def cpu_vector_addition_single_thread(n):
    a = np.random.rand(n)
    b = np.random.rand(n)
    c = np.empty_like(a)

    vector_add_single_thread(a, b, c)
```

Figure 5: Addition of two vectors sequentially in Python

When it comes to parallelizing the addition of two vectors in Python, we can approach this in two main ways:

- Vanilla Python with `concurrent.futures.ProcessPoolExecutor`, `Thread`, or any other class that facilitates parallelization, see fig. 6

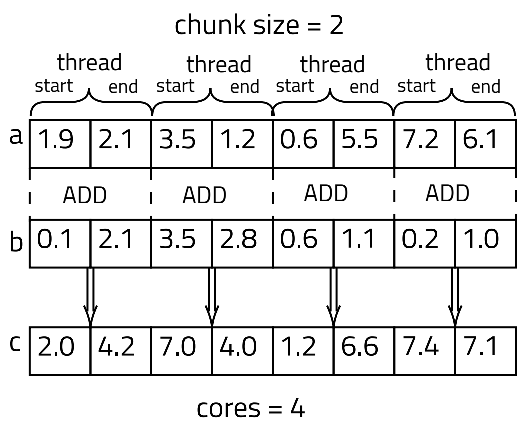

Figure 7: Add in parallel example

- Via external libraries such as Numba, see fig. 8

```python
import multiprocessing as mp

def vector_addition_chunk(start, end, a, b, c):
    for i in range(start, end):
        c[i] = a[i] + b[i]

def cpu_vector_addition_vanilla_python(n):
    # Get the number of available CPU cores - 24 on my machine
    num_cores = mp.cpu_count()


    # Generate two large vectors
    a = np.random.rand(n)
    b = np.random.rand(n)
    c = np.empty_like(a)

    chunk_size = len(a) // num_cores
    threads = []

    for i in range(num_cores):
        start = i * chunk_size
        end = (i + 1) * chunk_size if i != num_cores - 1 else len(a)
        thread = Thread(target=vector_addition_chunk, args=(start, end, a, b, c))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
```

Figure 6: Addition of two vectors parallel (CPU) in vanilla Python

The algorithm shown in Fig. 6 can be illustrated by the example in Fig. 7. For a 4-core CPU and vectors of size 8, the vector is divided into equal chunks of 2 elements, which are then computed in parallel.

```python
from numba import njit, prange

@njit(parallel=True) # <- Run parallel on CPU
def vector_add_multi_thread(a, b, c):
    for i in prange(a.size):
        c[i] = a[i] + b[i]

def cpu_vector_addition_multithread_numba(n):
    a = np.random.rand(n)
    b = np.random.rand(n)
    c = np.empty_like(a)

    vector_add_multi_thread(a, b, c)
```

Figure 8: Addition of two vectors parallel (CPU) in vanilla Python

In Table 1, we can see the average times of 10 runs for each implementation of the vector addition: CPU sequential, parallel vanilla, and parallel NUMBA.

		Structure Size				Speedup (Sequential / Parallel)			
Nr. of runs	Impl	10^5	10^6	10^7	10^8	10^5	10^6	10^7	10^8
10	Sequential	0.017300 seconds	0.148800 seconds	1.195475 seconds	11.835399 seconds				
10	Parallel Vanilla	0.020700 seconds	0.123999 seconds	1.201112 seconds	11.910937 seconds	0.8367x	1.2006x	0.9953x	0.9937x
10	Parallel Numba	0.022700 seconds	0.000701 seconds	0.004704 seconds	0.044202 seconds	0.7621x	212.411x	254.135x	267.698x

Table 1: Results of CPU Sequential/Parallel runs and their speedups.

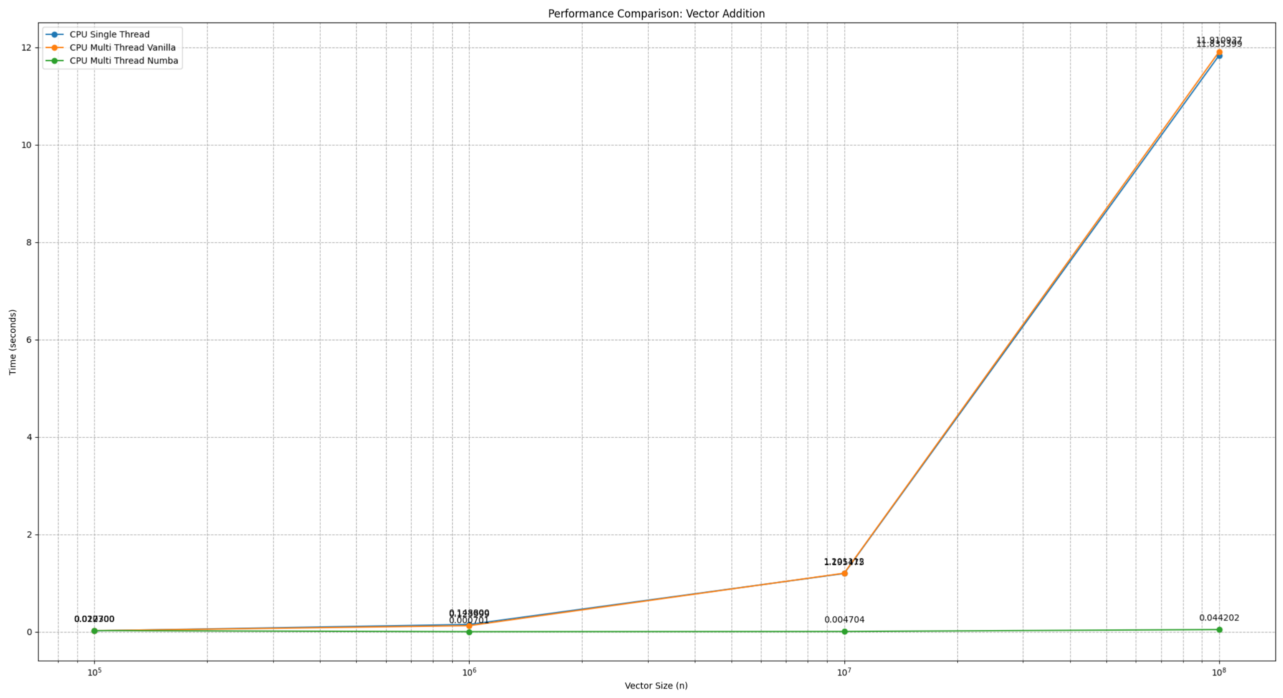

Figure 9: Matplotlib plot of the CPU runs

In Fig. 9 and Table 1, we can clearly observe that the sequential implementation and the vanilla parallel implementation have similar times, while the NUMBA implementation has significantly better times (lower is better). This happens because vanilla Python has overhead due to its layers and abstractions, whereas NUMBA uses highly optimized machine code, which runs directly on the CPU.

5.1.3 Python GPU Implementation

```python
def gpu_vector_addition_cupy_custom_kernel(n):
    a = cp.random.rand(n)
    b = cp.random.rand(n)

    # Define the custom CUDA kernel for vector addition
    vector_addition_kernel = cp.RawKernel(r'''
    extern "C" __global__
    void vector_addition(const float* a, const float* b, float* c, const int n) {
        int i = threadIdx.x;
        if (i < n) {
            c[i] = a[i] + b[i];
        }
    }
    ''', 'vector_addition')

    # Allocate memory for the result vector
    c = cp.empty_like(a)

    # Determine the number of threads and blocks - Query device properties
    device = cp.cuda.Device()
    max_threads_per_block = device.attributes['MaxThreadsPerBlock']
    max_blocks_per_grid = device.attributes['MaxGridDimX']

    blocks_per_grid = (n + max_threads_per_block - 1) // max_blocks_per_grid

    # Launch the custom kernel
    vector_addition_kernel((blocks_per_grid,), (max_threads_per_block,), (a, b, c,
    ↪  n))

    # Synchronize the device to ensure all operations are finished
    cp.cuda.Device().synchronize()
```

Figure 10: Addition of two vectors parallel (GPU) in CuPy with custom Kernel

CuPy library allows us to write custom kernels directly in C/C++, so the example from Fig. 10 is a direct representation of the code from Nvidia documentation in Fig. 4, except that in this code, instead of one block, the entire GPU is utilized.

We can also have a highly simplified version in CuPy. Since we work with CuPy data structures, we can simply use the add method, and the operations will be done in parallel, as seen in Fig. 11.

```python
def gpu_vector_addition_cupy(n):
    a = cp.random.rand(n)
    b = cp.random.rand(n)

    cp.add(a, b, c)

    # Synchronize the device to ensure all operations are finished
    cp.cuda.Device().synchronize()
```

Figure 11: Addition of two vectors parallel (GPU) in CuPy

Lastly, Numba also offers an annotation (`@cuda.jit`) for CUDA parallelization besides CPU parallelization (with the annotation `@njit(parallel=True)`). In Fig. 12, we can see the implementation.

```python
from numba import cuda
import numpy as np

@cuda.jit
def vector_add_gpu(a, b, c):
    idx = cuda.grid(1)
    if idx < a.size:
        c[idx] = a[idx] + b[idx]

def gpu_vector_addition_numba(n):
    a = np.random.rand(n)
    b = np.random.rand(n)

    # Allocate GPU memory and copy data to GPU
    a_device = cuda.to_device(a)
    b_device = cuda.to_device(b)
    c_device = cuda.device_array_like(a)

    # Configure the blocks and threads per block
    threads_per_block = 256
    blocks_per_grid = (a.size + (threads_per_block - 1)) // threads_per_block

    vector_add_gpu[blocks_per_grid, threads_per_block](a_device, b_device,
    ↪  c_device)
    cuda.synchronize()
```

Figure 12: Addition of two vectors in parallel (GPU) using NUMBA

In Table 2 and Fig. 13, we see comparisons between all implementations. Fig. 14 shows a zoomed-in version with only the parallel implementations in matplotlib. It is evident that the parallel versions are faster than the sequential versions. However, the parallel CPU implementation tends to slow down at a vector size of 10^8. An outlier is the method implemented in CuPy with the built-in add function for arrays. It appears that the custom kernel version performs much better than CuPy with add method, possibly due to overhead down the class hierarchy. This could be an interesting research topic to explore the implementation details. Another interesting aspect to analyze is the consistent speed increase at array sizes of 10^6, which could be due to special allocation (better cache utilization).

		Structure Size				Speedup (Sequential / Parallel)			
Nr. of runs	Impl	10^5	10^6	10^7	10^8	10^5	10^6	10^7	10^8
10	Sequential	0.0166047100	0.1429307300	1.1719473900	11.8722801800				
10	CPU Numba	0.0246411900	0.0005784900	0.0045634300	0.0459598300	0.673x	247.075x	256.812x	258.318x
10	GPU Numba	0.0056760400	0.0002938500	0.0150725600	0.0219791200	2.925x	486.407x	77.753x	540.161x
10	GPU CuPy	0.0000834700	0.0000902200	0.0326063800	0.2784807300	198.930x	1584.246x	35.942x	42.632x
10	GPU Custom Kernel	0.0000738800	0.0000436800	0.0229561000	0.0124657700	224.752x	3272.22x	51.051x	952.390

Table 2: Comparison between CPU Seq., CPU parallel, GPU parallel

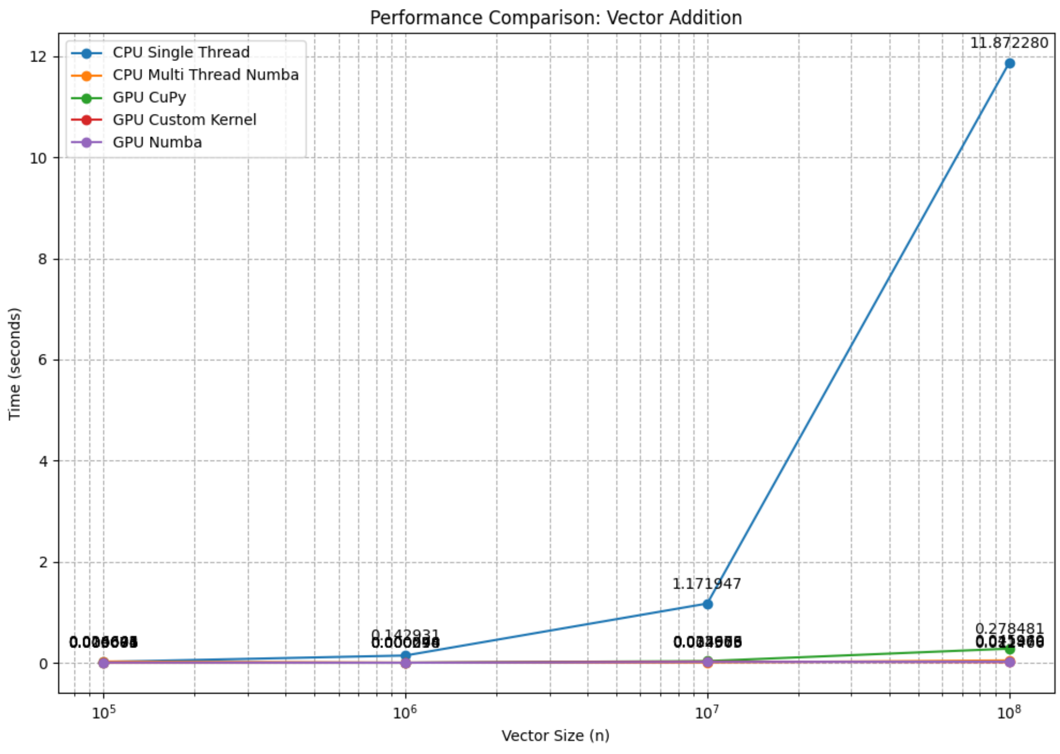

Figure 13: Matplotlib comparasions between all implementations

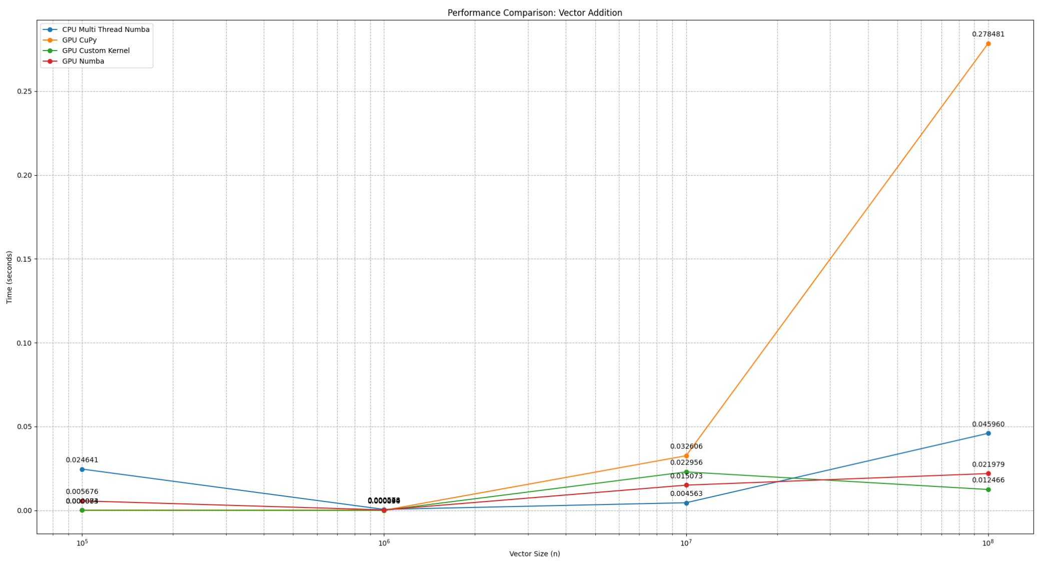

Figure 14: Matplotlib comparasions between all parallel implementations

5.2 Addition of Two Matrices

5.2.1 C++ CUDA implementation

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Figure 15: Addition of two matrices with CUDA in C++
Source: Nvidia documentation https://docs.nvidia.com/cuda

Additionally we can extend the code from Fig. 15 to handle multiple blocks as seen in (Fig. 16).
See more at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-block-clusters

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Figure 16: Addition of two matrices with CUDA in C++
Source: Nvidia documentation https://docs.nvidia.com/cuda

- **int i = blockIdx.x * blockDim.x + threadIdx.x;**: This calculates the row index for the current thread. **blockIdx.x** is the block index in the x-dimension, **blockDim.x**is the number of threads per block in the x-dimension, and **threadIdx.x** is the thread index within the block.
- **int j = blockIdx.y * blockDim.y + threadIdx.y;**: This calculates the column index for the current thread. **blockIdx.y** is the block index in the y-dimension, **blockDim.y** is the number of threads per block in the y-dimension, and **threadIdx.y** is the thread index within the block.
- **if (i < N && j < N)**: This condition ensures that the thread indices are within the bounds of the matrix. This is important because the grid of threads may be larger than the matrix.

5.3 Julia Sets