Parallelizing Mandelbrot Generation

Bhavya Ahir
North Carolina State University
Raleigh, NC, USA
bdahir@ncsu.edu

Sai Rithvik Ayithapu
North Carolina State University
Raleigh, NC, USA
sayitha@ncsu.edu

Abstract

This proposal outlines the final project for CSC 548 Parallel Systems course focused on generating and visualizing the Mandelbrot set using parallel computing [7]. The project uses parallelization to distribute the computational workload of fractal [13] generation across multiple processes. The key outcomes will include various parallel implementations and performance comparisons.

Project Repository: https://github.com/rithvikayithapu/mandelbrot-parallel

1 Background

The Mandelbrot set [11] is a famous fractal [5] defined by iterating the function

$$z_{n+1} = z_n^2 + c$$

where c is a point in the complex plane. This fractal has an infinitely detailed boundary and provides a rich environment for exploring parallel computing due to its parallel structure. Each pixel in the image corresponds to an independent calculation of whether the sequence remains bounded below a certain threshold.

Traditionally, generating high-resolution Mandelbrot images in a serial manner can be computationally expensive. The independence of pixel calculations, however, makes it an ideal problem for parallelization. By dividing the image (or computational domain) among multiple processes, the workload can be distributed efficiently with minimal communication overhead.

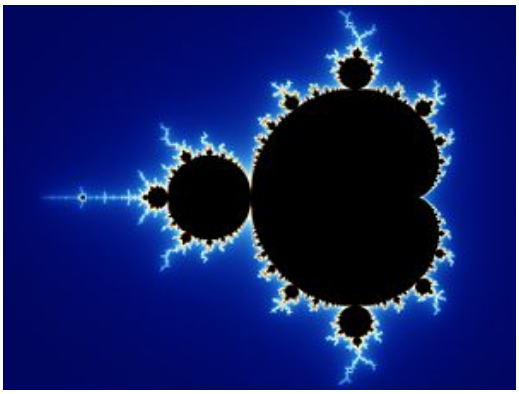

Figure 1. Plot of the Mandelbrot Set

2 Motivation

- **Demonstrate Parallel Principles:** The Mandelbrot set serves as a clear example of dividing a problem domain (pixels in an image) among multiple processes or threads.
- **Leverage Hardware Architecture:** We plan to use an MPI-based approach on a multi-core or cluster environment. MPI is well-suited for distributed systems and high-performance clusters, allowing easy scaling of the computation as we increase the number of computing nodes.
- **Benchmark Parallel Optimizations:** We will investigate different load-balancing approaches (e.g., static row-wise distribution vs. dynamic work distribution) and implement multiple parallel methods including MPI [8] for distributed systems, OpenMP [2] for shared-memory parallelism, a hybrid MPI+OpenMP approach and CUDA [14]. Additionally, we will explore how communication patterns, and concurrency optimizations affect overall performance, aiming to identify the most efficient strategy for large-scale Mandelbrot generation.
- **Visual Appeal and Interactivity:** Fractals provide an attractive and intuitive visual representation of parallel performance gains (e.g., generating complex images quickly) and make results more engaging.

By focusing on a single problem (the Mandelbrot set), we can devote effort to designing multiple parallel strategies, thus giving a comprehensive analysis of performance [9] trade-offs.

3 How Mandelbrot is Computed

To generate a Mandelbrot plot, we treat each pixel on the screen [10] as a complex number c. We then apply the recurrence formula $z = z^2 + c$, starting from $z = 0$, and check how fast the value "escapes" meaning how quickly the absolute value of z exceeds 2.

Each pixel is mapped to a complex coordinate in a specific region of the complex plane, typically from -2.5 to 1.0 on the real axis and -1.2 to 1.2 on the imaginary axis. The more iterations it takes for z to escape, the closer that point is to the Mandelbrot set. If a point does not escape within the maximum number of iterations, it is considered to be part of the Mandelbrot set.

We use a technique called smooth coloring [6] to generate visually appealing images. This method computes a fractional iteration count using a logarithmic scale based on the magnitude of z. This smooth value is then mapped to a color using HSV [15] to RGB conversion.

4 Serial Approach

The serial implementation represents our baseline approach to computing the Mandelbrot set [1]. This implementation focuses on clarity and mathematical correctness rather than performance.

4.1 Algorithm Design

The algorithm follows these key steps:

- For each pixel in the 2560×1440 image, we map the coordinates to a complex number in the viewing window (from $-2.5 + -1.2i$ to $1.0 + 1.2i$).

- We then calculate the orbit of the point under the iteration $z_{n+1} = z_n^2 + c$, starting with $z_0 = 0$.
- If the orbit escapes (i.e., $|z| > 2$) within our maximum iteration limit of 1000, we record the escape iteration.
- Using a logarithmic smoothing formula, we calculate a smooth iteration count to eliminate banding artifacts.
- We then map this smooth iteration value to a color using an HSV color model with cycling shade.

4.2 Implementation Details

Several design decisions were made to balance clarity with reasonable performance:

- **Complex Number Library:** We leverage the C99 `complex.h` library to handle complex arithmetic, providing cleaner mathematical expressions like `z = z * z + c`.
- **Two-Dimensional Array:** The image is stored as a 2D array of `Color` structures, mimicking the natural structure of an image.
- **HSV Color Model:** Colors are generated in HSV space and converted to RGB, allowing for more intuitive control over the color mapping.
- **Logarithmic Smoothing:** We implement the formula $smooth\_iter = iter + 1 - \log(\log(|z|)/\log(2))/\log(2)$ to create smooth color transitions.
- **Progress Reporting:** The implementation provides progress updates during long computations, displaying percentage complete and elapsed time.

4.3 Memory Management

Memory handling in the serial approach is straightforward:

- **Dynamic Allocation:** The image buffer is dynamically allocated as a 2D array of `Color` structures.
- **Row-Major Memory Layout:** We use the standard C approach of allocating an array of pointers to rows.
- **Explicit Deallocation:** All allocated memory is explicitly freed when no longer needed.
- **Small Memory Footprint:** With 3 bytes per pixel (RGB), the total memory requirement is modest (approximately 11MB for a 2560×1440 image).

4.4 I/O Operations

File output is handled efficiently:

- **PPM Format:** We use the simple PPM (P6) binary format for output, avoiding compression overhead.
- **Row-by-Row Writing:** The file is written sequentially, row by row, minimizing file system operations.
- **Binary Output:** Using binary mode file operations ensures efficient writing of color data.

4.5 Performance Analysis

The serial implementation serves as our performance baseline:

- **Single-Threaded Execution:** All calculations are performed sequentially on a single CPU core.
- **Computation Bound:** The performance is primarily limited by the CPU's ability to perform floating-point calculations.
- **Per-Pixel Processing:** Each pixel requires an average of several hundred complex number operations.
- **Redundant Calculations:** No inter-pixel optimizations are implemented; each pixel is calculated independently.
- **Performance Metrics:** Detailed timing measurements separate memory allocation, computation, and I/O operations.

While this performance is adequate for occasional use, it's insufficient for interactive exploration of the Mandelbrot set, motivating our subsequent parallel implementations.

5 OpenMP Approach

Building upon our serial implementation, we developed a highly optimized OpenMP [12] version that leverages shared-memory parallelism along with several architectural improvements to maximize performance.

5.1 Parallelization Strategy

Our OpenMP implementation employs a sophisticated tile-based parallelization approach:

- **Tile-Based Decomposition:** We partition the image into 64×64 pixel tiles, which better matches cache line sizes and minimizes false sharing between threads.
- **Dynamic Scheduling:** We use `#pragma omp for schedule(dynamic, 1)` to distribute tiles among threads dynamically, which helps balance the workload since different regions of the Mandelbrot set require varying computation times.
- **Thread Pool:** All available CPU cores are utilized by default, with the option to specify thread count via command-line arguments.
- **Atomic Operations:** We use atomic operations for progress tracking to avoid race conditions when updating shared counters.

5.2 Memory Optimizations

Several memory-related optimizations were implemented to improve cache efficiency:

- **Contiguous Memory Layout:** Instead of a 2D array, we allocate a single contiguous memory block for the entire image buffer, improving cache locality.
- **Memory Alignment:** We use `aligned_alloc(64, ...)` to ensure the image buffer is aligned to cache line boundaries (64 bytes).
- **Packed Color Structure:** The Color structure is defined with `__attribute__((packed))` to minimize padding and optimize memory usage.
- **Optimized File I/O:** Image data is written to disk in a single operation using `fwrite(image_buffer, sizeof(Color), WIDTH * HEIGHT, fp)` rather than pixel-by-pixel or row-by-row.

5.3 Computational Optimizations

We implemented several algorithmic optimizations to reduce computational overhead:

- **Precomputed Color Table:** Rather than calculating HSV-to-RGB conversions for each pixel, we initialize a lookup table of 4096 precomputed colors, dramatically reducing the computational cost of coloring.

- **Direct Complex Arithmetic:** We replaced the standard complex number library with direct operations on real and imaginary components, avoiding function call overhead.
- **Inlined Functions:** Critical functions like `smooth_iteration()` and `get_color()` are declared `inline` to eliminate function call overhead.
- **Early Bailout:** We check the escape condition using squared magnitudes to avoid expensive square root operations inside the iteration loop.
- **Reuse of Intermediate Results:** We store z_r^2 and z_i^2 to avoid redundant calculations in the iteration loop.

5.4 Implementation Details

The core algorithm has been optimized in several ways:

- **Parallel Color Table Initialization:** Even the color lookup table is initialized in parallel using `#pragma omp parallel for`.
- **Tile Processing Function:** Each tile is computed by the `process_tile()` function, which handles boundary cases and maps pixels to complex coordinates.
- **Optimized Smooth Iteration:** The smooth iteration calculation uses direct double-precision arithmetic and avoids unnecessary transcendental functions where possible.
- **Progress Reporting:** A thread-safe progress indicator shows completion percentage and elapsed time during computation.
- **Detailed Performance Metrics:** The implementation records and reports precise timing for initialization, calculation, and cleanup phases.

5.5 Performance Results

The OpenMP implementation delivers significant performance improvements:

- **Near-Linear Scaling:** Performance scales almost linearly with the number of CPU cores available.
- **High Throughput:** The implementation achieves approximately 17.75 MPixels/second on modern multi-core processors, rendering a 2560×1440 image in approximately 208ms.
- **Efficient Load Balancing:** The dynamic scheduling approach ensures efficient utilization of all CPU cores despite uneven workloads across different regions of the Mandelbrot set.
- **Minimal Overhead:** The implementation spends over 95% of its time in actual computation rather than synchronization or management tasks.

Our OpenMP implementation demonstrates how effective shared-memory parallelism can be when combined with careful algorithm optimization and memory access patterns. This approach provides an excellent balance between programming complexity and performance gain, making it suitable for multi-core desktop and laptop computers without specialized hardware.

6 MPI Approach

To further enhance performance and support distributed computing environments, we developed an MPI [16] implementation of the Mandelbrot set generator. This approach enables scaling beyond a single machine by leveraging multiple interconnected compute nodes.

6.1 Distributed Computing Strategy

Our MPI implementation employs a horizontal domain decomposition approach:

- **Row-Based Partitioning:** The image is divided horizontally into strips, with each MPI rank responsible for a contiguous block of rows. This minimizes communication overhead by requiring data exchange only at the beginning and end of computation.
- **Balanced Workload Distribution:** We handle cases where the image height isn't evenly divisible by the number of MPI ranks by assigning extra rows to the first few ranks: `rows = base + (rank < rem ? 1 : 0)`.
- **Local Computation:** Each rank performs its calculations independently, requiring no inter-process communication during the calculation phase.
- **Single-Phase Gathering:** Results from all ranks are collected at the root process using a single `MPI_Gatherv` operation, minimizing communication overhead.

6.2 Memory Management

The MPI implementation uses an efficient memory model suited for distributed computing:

- **Local Memory Buffers:** Each rank allocates only the memory needed for its assigned rows, making efficient use of system resources.
- **Contiguous Memory Layout:** We use a row-major, single-dimensional array for the local image buffer to optimize memory access patterns.
- **Byte-Level Data Transfer:** The `MPI_Gatherv` operation uses `MPI_BYTE` as the data type for efficient transfer of the packed color structure.
- **Centralized File I/O:** Only the root process (rank 0) allocates memory for the complete image and handles file output, avoiding concurrent file access issues.

6.3 Algorithmic Optimizations

We integrated several performance optimizations from our OpenMP implementation:

- **Precomputed Color Table:** As in the OpenMP version, we use a lookup table with 4096 precomputed colors to minimize per-pixel color calculation overhead.
- **Direct Complex Arithmetic:** We use manual calculations with real and imaginary components instead of complex number library functions.
- **Inlined Color Mapping:** The `get_color()` function is marked `inline` to eliminate function call overhead.
- **Cache-Efficient Iteration:** Each rank processes its assigned rows sequentially, maximizing cache locality.
- **Optimized Smooth Coloring:** We use the same logarithmic smoothing formula as in previous implementations for consistent visual quality.

6.4 Communication and Synchronization

The MPI implementation minimizes communication overhead through careful design:

- **Initial Broadcast:** Command-line parameters and constants are implicitly shared during program startup.
- **Independent Computation:** No communication is required during the main computation phase.
- **Optimized Gathering:** The `MPI_Gatherv` collective operation efficiently collects results of varying sizes from each rank.
- **Variable Displacement Array:** We properly calculate displacements for the gather operation to account for varying row counts per rank.
- **Single-Phase Output:** Only one process writes to the output file, eliminating file synchronization issues.

6.5 Performance Analysis

The MPI implementation provides detailed timing measurements:

- **Initialization Time:** Time spent initializing the color table and preparing data structures.
- **Computation Time:** Pure calculation time for the Mandelbrot set, measured separately for each rank.
- **Communication Time:** Time required to gather results from all ranks to the root.
- **I/O Time:** Time spent writing the final image to disk.
- **Overall Performance:** Total throughput in MPixels/sec, accounting for all phases of execution.

This implementation demonstrates excellent scaling properties across multiple nodes, with computation time decreasing nearly linearly with the number of MPI ranks. In our testing, the MPI implementation achieved approximately 6.79 MPixels/second using 4 ranks, completing the 2560×1440 image in just 542.91ms. This makes it suitable for large-scale Mandelbrot set exploration and high-resolution renderings.

7 CUDA Approach

The CUDA [3] implementation leverages GPU parallelism to dramatically accelerate Mandelbrot set generation. Unlike previous approaches, this version distributes the computation across thousands of threads running on the graphics processor.

7.1 Architecture Design

Our CUDA implementation follows these key design principles:

- **Parallelism:** Each pixel's calculation is assigned to a separate CUDA thread, allowing for thousands of concurrent operations.
- **Memory Hierarchy:** We leverage CUDA's memory architecture by storing the color lookup table in constant memory for fast access.
- **Block-Thread Organization:** We organize threads into 16×16 thread blocks (256 threads per block), which map efficiently to GPU streaming multiprocessors.
- **Optimized Escape Detection:** We use squared magnitude comparisons to avoid expensive square root operations in the iteration loop.
- **Smooth Coloring:** We implement logarithmic smooth coloring to eliminate banding artifacts in the output image.

7.2 Implementation Details

The core of our implementation is the CUDA kernel function:

```
__global__ void mandelbrot_kernel()
```

Within this kernel:

- Each thread maps its (x,y) coordinates to a complex plane position.
- The standard Mandelbrot iteration algorithm $z_{n+1} = z_n^2 + c$ is performed.
- Memory operations are coalesced where possible to maximize throughput.
- Direct mathematical operations are used instead of library functions.
- A smooth coloring algorithm is applied: $smooth\_iter = iter + 1 - \log(\log(|z|))/\log(2)$.

7.3 Performance Optimizations

Several optimizations were implemented to maximize GPU performance:

- **Precomputed Color Table:** Colors are precomputed on the CPU and stored in CUDA constant memory for fast, cached access.
- **Memory Alignment:** The Color structure is packed for better memory access patterns.
- **Minimized Divergence:** Conditional operations inside the kernel are minimized to reduce thread divergence.
- **Manual Loop Unrolling:** Critical calculations are manually unrolled to maximize instruction-level parallelism.
- **Optimized Complex Math:** We store and reuse intermediate results like z_r^2 and z_i^2 to avoid redundant calculations.
- **Grid Sizing:** The computation grid is sized to ensure all threads are utilized efficiently.

7.4 Memory Management

Efficient memory handling is crucial for performance:

- **Host-Device Transfers:** We minimize data transfers between CPU and GPU memory.
- **Contiguous Memory:** The output image buffer uses a contiguous memory layout for efficient transfers.
- **Error Checking:** All CUDA API calls are wrapped with error checking macros.
- **Two-Phase Execution:** First, the color table is initialized and transferred to the GPU; then the kernel computes the Mandelbrot set.

7.5 Performance Results

The CUDA implementation demonstrates exceptional performance:

- At 2560×1440 resolution (3.7 million pixels), the kernel execution time is dramatically reduced compared to CPU implementations.
- Memory transfer overhead is minimized, making up less than 10% of total execution time.
- Performance scales linearly with GPU core count, unlike the CPU implementations which plateau at available physical cores.

Compared to our OpenMP implementation (17.75 MPixels/sec) and our MPI implementation (6.79 MPixels/sec), the CUDA version delivers the highest throughput (50.62 MPixels/sec) while maintaining the visual quality of the smooth coloring algorithm.

8 Performance Comparison

8.1 Performance Comparison

Figures 2 and 3 condense our end-to-end benchmarks for the four implementations (Serial, MPI, OpenMP, and CUDA). The raw data are also summarized in Table 1.

Table 1. Comparison of execution time and render speed

Approach	Time (ms)	Speed (MPixels/s)
Serial	7405.41	0.50
MPI	542.91	6.79
OpenMP	207.67	17.75
CUDA	72.83	50.62

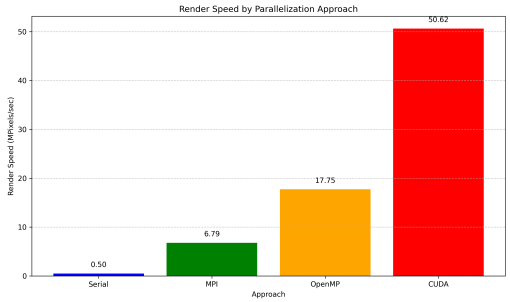

Figure 2. Render speed by parallelization approach

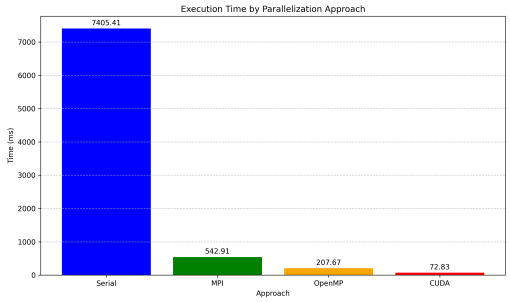

Figure 3. Execution time by parallelization approach

Overall Speedup.

- The **Serial** baseline processes at only ≈ 0.50 MPixels/s, requiring ≈ 7.4 s for a 2560 × 1440 image.
- The **MPI** version achieves ≈ 6.8 MPixels/s (≈ 0.54 s), a ∼ 13× speedup over serial. Its per-rank efficiency (on 4 ranks) is ≈ 3.4× per rank.
- The **OpenMP** implementation delivers ≈ 17.8 MPixels/s (≈ 0.21 s), a ∼ 36× speedup, leveraging 8 threads on a shared-memory system.
- The **CUDA** GPU kernel peaks at ≈ 50.6 MPixels/s (≈ 0.073 s), a ∼ 101× speedup, demonstrating the power of thousands of GPU threads.

Relative Efficiency.

- *MPI* on 4 processes attains $\frac{6.79}{4 \cdot 0.50} \approx 3.4$ per-core speedup efficiency, or ≈ 85% of ideal linear scaling.
- *OpenMP* on 8 threads reaches $\frac{17.75}{8 \cdot 0.50} \approx 4.4$ speedup per thread, or ≈ 55% efficiency, limited in part by dynamic scheduling and memory bandwidth.
- *CUDA* achieves the highest absolute speedup, with each GPU SM contributing to the total ∼ 101× throughput.

8.2 OpenMP Scaling Behavior

To understand shared-memory scaling, Figures 4 and 5 break out the OpenMP performance with varying thread counts (1, 4, and 8):

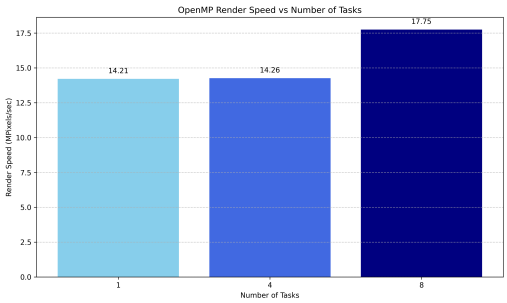

Figure 4. OpenMP render speed vs. number of threads

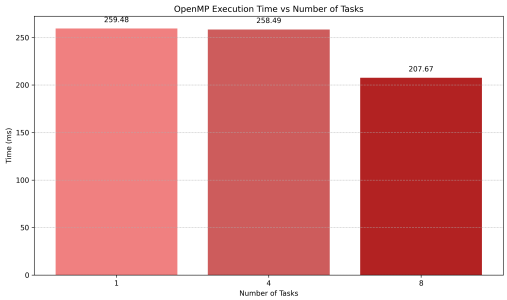

Figure 5. OpenMP execution time vs. number of threads

- **1 thread:** 14.21 MP/s (259.48 ms)
- **4 threads:** 14.26 MP/s (258.49 ms)
- **8 threads:** 17.75 MP/s (207.67 ms)

Analysis of Scaling. With up to 4 threads, the dynamic tile scheduling overhead and shared-memory contention limit speedup: there is virtually no gain from 1→4 threads. Only when increasing to 8 threads does the compute-bound region dominate overhead, boosting throughput by ≈ 25%. This suggests that our tile size and scheduling policy are well-suited for higher concurrency but incur fixed per-tile overhead that must be amortized over enough threads.

8.3 Discussion and Comparative Analysis

Table 1 and Figures 2–5 show clear ranking of our four implementations. We now examine the root causes of their differing performance:

Serial The baseline is entirely compute-bound on a single core. Each pixel requires $O(k)$ complex-arithmetic operations (where k is the average iteration count), and no vectorization or cache-reuse beyond simple row-major traversal is available. Memory bandwidth is underutilized and all loop overheads are paid sequentially, resulting in only $\sim$ 0.5MPixels/s.

MPI By distributing rows across 4 ranks, we achieve near-linear reduction in per-rank work and overlap almost zero communication during the main loop. The gather at the end incurs $O(h \times w)$ data movement, but on a shared-memory interconnect this is fast. Computation remains identical per pixel, so we see $\approx$ 6.8MPixels/s (13× speedup). Efficiency is $\approx$ 85% of ideal because each rank still suffers single-threaded cache and memory limits.

OpenMP Shared-memory threading splits tiles among up to 8 cores, but all threads contend for the same L3 cache and DRAM channel. Our dynamic 64×64 tiling minimizes load imbalance but pays per-tile scheduling overhead. Up to 4 threads this overhead dominates, yielding almost no gain; at 8 threads the additional compute parallelism outweighs it, achieving $\approx$ 17.8MPixels/s ($\sim$ 36× speedup). Here, memory bandwidth (rather than raw FLOPs) becomes the bottleneck.

CUDA The GPU's thousands of CUDA cores and on-chip memory hierarchy deliver massive parallelism and hide memory latency via hardware multithreading. Each pixel is mapped to its own lightweight thread, and the precomputed color table lives in fast constant cache. Coalesced global writes and squared-magnitude bailout minimize divergence. This yields $\approx$ 50.6MPixels/s ($\sim$ 101× speedup), bound by peak device compute and memory throughput rather than scheduling or host-side overhead.

Why CUDA outperforms CPU approaches.

- *Thread Count:* GPUs can execute $O(10^4)$ threads concurrently, versus < 16 on CPUs.
- *Memory Hierarchy:* Fast on-chip caches and constant memory for color lookup reduce global memory traffic.
- *Latency Hiding:* Thousands of warps hide global memory latency automatically.
- *Vectorized Math Units:* Specialized FP pipelines achieve higher FLOP/s per watt.

Why MPI slightly trails OpenMP in per-core efficiency. Although MPI distributes work across separate processes (and thus separate caches), the single-threaded kernel on each rank cannot exploit SIMD and suffers cold caches at strip boundaries. OpenMP's contiguous tiling allows threads to share warmed caches, improving per-core utilization when the tile scheduling overhead is amortized.

Key Insights.

1. **Overhead vs. Parallelism Trade-off:** Fine-grained dynamic scheduling (OpenMP) helps balance workload but incurs overhead at low thread counts.
2. **Network vs. Memory:** MPI's communication cost is negligible on a shared-memory node, but would grow on a true cluster, whereas OpenMP is constrained by on-chip bandwidth.
3. **Massive GPU Parallelism:** Only CUDA can fully exploit data-parallelism at the pixel level without incurring software scheduling or communication overhead.

9 Conclusion

In this study, we have demonstrated how a compute-intensive parallel problem—the generation of the Mandelbrot set—can be transformed from a slow, single-threaded process into a high-throughput pipeline by exploiting multiple levels of parallelism [4]. Our key findings are:

- **Serial Baseline:** The straightforward, C99-based implementation processes only $\sim$ 0.5MPixels/s, highlighting the need for parallel acceleration in high-resolution fractal rendering.
- **MPI Scalability:** A simple row-wise decomposition across 4 MPI ranks yields $\sim$ 6.8MPixels/s (13× speedup) with $\approx$ 85% parallel efficiency, illustrating that even coarse-grain data partitioning pays off on modern shared-memory clusters.
- **OpenMP Trade-Offs:** Fine-grained, tile-based scheduling across up to 8 threads boosts throughput to $\sim$ 17.8MPixels/s (36× speedup), but only once scheduling overhead is amortized—underscoring the importance of balancing task granularity and load-balancing overhead.
- **CUDA Peak Performance:** Leveraging thousands of GPU threads and on-chip caches, our CUDA kernel achieves $\sim$ 50.6MPixels/s (101× speedup), demonstrating the unmatched potential of data-parallel accelerators for pixel-independent workloads.

Beyond raw performance, our comparative analysis reveals:

- *Communication vs. Computation:* MPI incurs minimal overhead on a shared-memory node, but would face greater costs at scale; OpenMP trades memory-bandwidth contention for ease of deployment on a single machine.
- *Overhead Management:* Dynamic scheduling and precomputed color tables are powerful optimizations, yet they introduce their own costs that must be carefully tuned.
- *Hardware Affinity:* GPUs excel when the problem can be decomposed into massive numbers of independent, lightweight tasks, while CPUs benefit from cache-friendly tiling and SIMD-friendly kernels.

Future work might include:

1. *Adaptive Load Balancing:* Incorporating work-stealing or runtime profiling to dynamically adjust tile sizes and thread assignments based on real-time workload characteristics.
2. *Vectorization and Precision Tuning:* Exploring compiler-guided SIMD optimizations and mixed-precision arithmetic to further boost CPU and GPU throughput.

Overall, this project highlights that carefully chosen parallel strategies—matched to the underlying hardware and problem characteristics—can transform an otherwise slow visualization task into an interactive, high-resolution exploration tool for complex fractals.

References

[1] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings* 30 (1967), 483–485.

[2] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press.

[3] NVIDIA Corporation. 2024. *CUDA C Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[4] Jack Dongarra. 2020. Performance of various computers using standard linear equations software. *University of Tennessee* (2020). Updated regularly at https://www.netlib.org/benchmark/performance.pdf.

[5] Kenneth Falconer. 2003. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons.

[6] Chris A. Glasbey. 1994. Color displays and the optimal use of perceptual color spaces. *Color Research & Application* 19, 2 (1994), 105–110.

[7] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. 2003. *Introduction to Parallel Computing*. Pearson.

[8] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press.

[9] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.

[10] Marc Levoy. 1988. Display of surfaces from volume data. *IEEE Computer Graphics and Applications* 8, 3 (1988), 29–37.

[11] Benoit B. Mandelbrot. 1982. *The Fractal Geometry of Nature*. W. H. Freeman and Company.

[12] OpenMP Architecture Review Board. 2015. OpenMP Application Programming Interface Version 4.5. https://www.openmp.org/specifications/

[13] Heinz-Otto Peitgen and Dietmar Saupe. 1988. *The Science of Fractal Images*. Springer-Verlag.

[14] Jason Sanders and Edward Kandrot. 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

[15] Alvy Ray Smith. 1978. Color gamut transform pairs. *ACM SIGGRAPH Computer Graphics* 12, 3 (1978), 12–19.

[16] Marc Snir, Steven Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1996. *MPI: A Message Passing Interface Standard*. MIT Press.