HESP Assignment 1

Parallel Computing with CUDA

RISHYAVANDHAN VENKATESAN - fi11maka
PRASHANTH GADWALA - yl34esew
AHEMAD DANIYAL - oq18afiw

May 11, 2025

Contents

1 Introduction

This report provides a comprehensive analysis of Assignment 1, which focuses on GPU programming using CUDA. The assignment consists of two main components:

- **Julia Set Generator:** A CUDA implementation that generates fractal images based on the Julia set mathematical concept.
- **Stream Benchmark:** A performance comparison between serial CPU execution, OpenMP parallel CPU execution, and CUDA GPU execution for a memory-bound streaming operation.

2 Julia Set Generator

2.1 Mathematical Background

The Julia set, named after French mathematician Gaston Julia, is a set of complex numbers that forms a fractal pattern when iteratively applying a specific function. For a complex function $f(z)$, the Julia set consists of values where the behavior of iterating f is chaotic.

In our implementation, we use the quadratic Julia set defined by:

$$f_c(z) = z^2 + c \tag{1}$$

where c is a complex constant parameter that determines the specific Julia set, and z is a complex number representing a point in the complex plane. The iteration starts with a point z_0 and repeatedly applies the function:

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

A point belongs to the Julia set if the sequence remains bounded when the function is repeatedly applied. In practice, we check if the magnitude of z_n exceeds a certain threshold (typically 2) after a specified number of iterations.

2.2 Implementation Analysis

The Julia set generator is implemented using CUDA to leverage the massively parallel architecture of GPUs. The core components include:

2.2.1 CUDA Kernel

The kernel function `juliaKernel` operates on individual pixels in parallel:

```
__global__ void juliaKernel(unsigned char* img, float cr, float ci) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= WIDTH || y >= HEIGHT)
        return;

    // Map pixel position to complex plane
    float zx = X_MIN + (X_MAX - X_MIN) * x / (WIDTH - 1);
    float zy = Y_MIN + (Y_MAX - Y_MIN) * y / (HEIGHT - 1);

    int iter = 0;
    float zx2 = zx * zx;
    float zy2 = zy * zy;

    // Standard Julia set iteration: z = z   + c
    while (iter < MAX_ITER && zx2 + zy2 < 4.0f) {
        float xtemp = zx2 - zy2 + cr;
        zy = 2 * zx * zy + ci;
        zx = xtemp;
        zx2 = zx * zx;
        zy2 = zy * zy;
        iter++;
    }

    // Calculate color based on iteration count
    int idx = 4 * (y * WIDTH + x);

    if (iter == MAX_ITER) {
        // Points in the set are black
        img[idx]     = 0;     // R
        img[idx + 1] = 0;     // G
        img[idx + 2] = 0;     // B
        img[idx + 3] = 255;   // Alpha
    } else {
        // Map iteration count to a smooth color
        float t = (float)iter / MAX_ITER;

        // RGB coloring (simple gradient)
        img[idx]     = (unsigned char)(255 * t);           // R
        img[idx + 1] = (unsigned char)(255 * (1.0f - t));  // G
        img[idx + 2] = (unsigned char)(128);               // B
        img[idx + 3] = 255;                                // Alpha
    }
}
```

Listing 1: Julia Set CUDA Kernel

This kernel works by:

1. Computing the global index of the thread, which corresponds to a pixel in the image.

2. Mapping the pixel coordinates to the complex plane using linear interpolation.
3. Iteratively applying the Julia set function $f_c(z) = z^2 + c$ until either:
    - The orbit exceeds a magnitude of 2 (indicating the point is not in the Julia set).
    - The maximum number of iterations is reached (suggesting the point may be in the Julia set).
4. Coloring the pixel based on the number of iterations performed, creating a visual representation of the fractal.

2.2.2 Memory Management

The implementation follows standard CUDA memory management practices:

```
// Allocate host memory for the image
std::vector<unsigned char> h_img(WIDTH * HEIGHT * 4, 0);

// Allocate device memory
unsigned char* d_img = nullptr;
cudaMalloc(&d_img, WIDTH * HEIGHT * 4);

// Initialize device memory to zero
cudaMemset(d_img, 0, WIDTH * HEIGHT * 4);

// ... [Kernel execution] ...

// Copy result back to host
cudaMemcpy(h_img.data(), d_img, WIDTH * HEIGHT * 4,
    cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_img);
```

Listing 2: Memory Management in Julia Set Implementation

This pattern encompasses:

- Allocating memory on the host (CPU)
- Allocating corresponding memory on the device (GPU)
- Initializing device memory
- Copying results from device to host after kernel execution
- Properly freeing device memory

2.2.3 Thread Organization

The kernel launches with a grid of thread blocks to process the entire image in parallel:

```
// Set up grid and block dimensions
dim3 blockDim(16, 16);
dim3 gridDim((WIDTH + blockDim.x - 1) / blockDim.x,
             (HEIGHT + blockDim.y - 1) / blockDim.y);

// Launch the kernel
juliaKernel<<<gridDim, blockDim>>>(d_img, cr, ci);
```

Listing 3: Thread Organization for Julia Set

The implementation uses a 2D grid of 2D thread blocks, which is a natural fit for image processing:

- Each thread block contains $16 \times 16 = 256$ threads
- The grid dimensions are calculated to ensure coverage of the entire image
- This structure provides efficient thread utilization and memory access patterns

2.3 Execution Process

The execution workflow is controlled by the `run.sh` script, which:

1. Compiles the necessary components (LodePNG library and the CUDA code)
2. Links the compiled objects into an executable
3. Runs the program with different Julia set parameters to generate various fractal images

Of particular interest is how it configures the CUDA compiler:

```
# For RTX 4050, the compute capability is 8.9 (Ada Lovelace architecture)
NVCC_FLAGS="--std=c++17 -arch=sm_89 -Xcompiler=-std=c++20,-fPIC -Wno-deprecated-gpu-targets -I$CUDA_INCLUDE"
```

Listing 4: CUDA Compilation Settings

This specifies:

- The target architecture (`sm_89` for Ada Lovelace GPUs)
- C++17 standard for CUDA code
- C++20 standard for host code
- Position-independent code generation

2.4 Julia Set Visualization

The program generates several Julia set images with different parameters:

Real Part (c_r)	**Imaginary Part (c_i)**	**Output File**
-0.8	0.2	julia_classic.png
0.285	0.01	julia_spiral1.png
-0.4	0.6	julia_swirl.png
-0.70176	-0.3842	julia_dense.png
0.355	0.355	julia_eye.png
-0.75	0.11	julia_nebula.png

Table 1: Julia Set Configurations Used in the Assignment

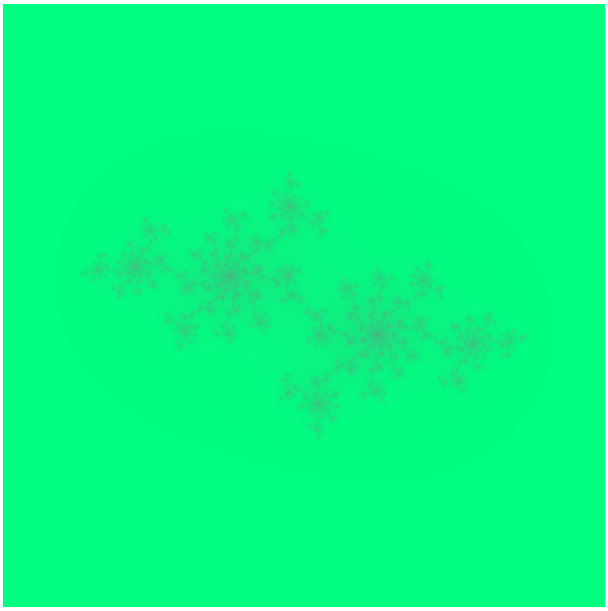

Figure 2: Julia Dense

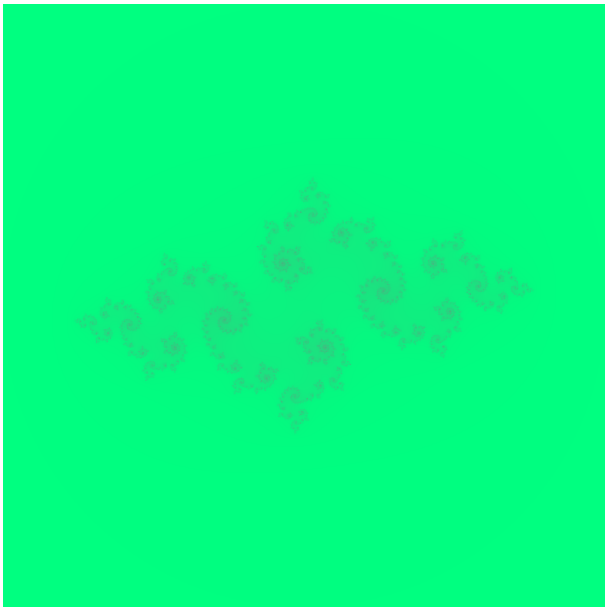

Figure 1: Julia Classic

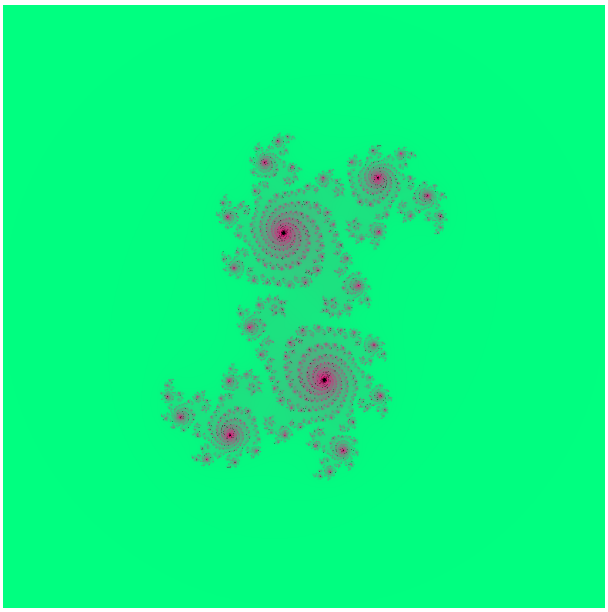

Figure 3: Julia Eye

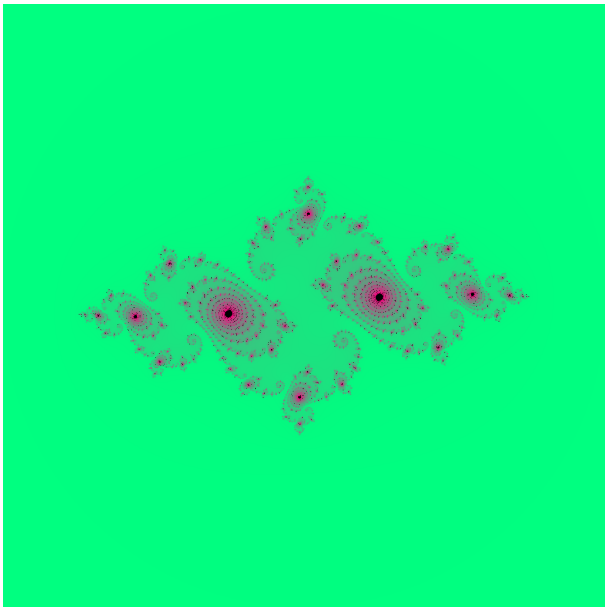

Figure 4: Julia Nebula

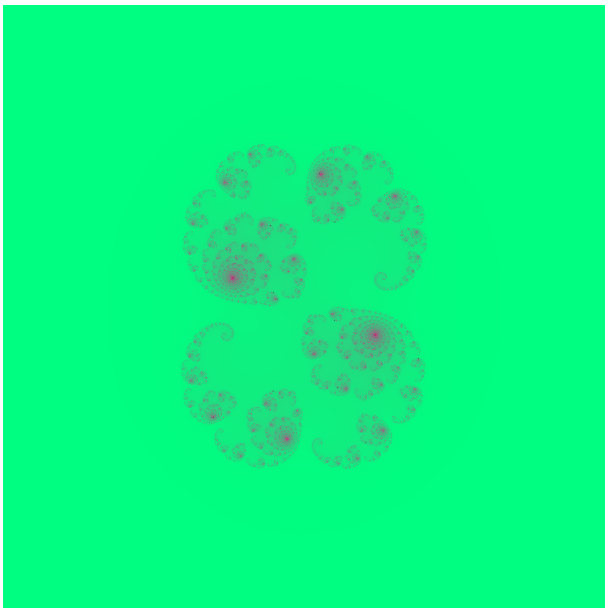

Figure 5: Julia Spiral

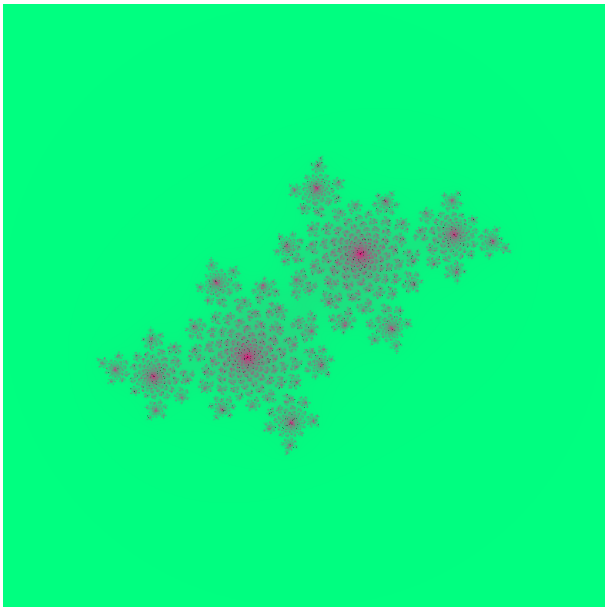

Figure 6: Julia Swirl

3 Stream Benchmark

3.1 Overview of the Stream Benchmark

The stream benchmark is a common tool for measuring memory bandwidth. It consists of simple operations that stress the memory subsystem rather than the computational units. In this assignment, we implement and compare three versions of a stream operation:

1. **Serial CPU Implementation:** Single-threaded execution on the CPU
2. **OpenMP CPU Implementation:** Parallel execution using OpenMP on the CPU
3. **CUDA GPU Implementation:** Parallel execution on the GPU using CUDA

3.2 Implementation Analysis

3.2.1 Operation Definition

The core operation performed in all implementations is:

$$\text{dest}[i] = \text{src}[i] + 1 \tag{3}$$

This simple addition is ideal for bandwidth testing because:

- It performs minimal computation (just one addition)
- It requires reading from one memory location and writing to another
- It doesn't reuse data, ensuring the operation is memory-bound

3.2.2 Serial CPU Implementation

The serial implementation is straightforward:

```
inline void stream(size_t nx, const double *__restrict__ src, double *
    __restrict__ dest) {
    for (int i = 0; i < nx; ++i)
        dest[i] = src[i] + 1;
}
```

Listing 5: Serial Stream Implementation

Key features:

- Uses `__restrict__` to inform the compiler that arrays don't overlap
- Sequentially processes all elements in the arrays
- Simple for-loop iteration pattern

3.2.3 OpenMP CPU Implementation

The OpenMP implementation parallelizes the loop using multiple CPU threads:

```
inline void stream(size_t nx, const double *__restrict__ src, double *
    __restrict__ dest) {
# pragma omp parallel for schedule (static)
    for (int i = 0; i < nx; ++i)
        dest[i] = src[i] + 1;
}
```

Listing 6: OpenMP Stream Implementation

Key features:

- Uses OpenMP's `#pragma omp parallel for` directive to parallelize the loop
- Employs static scheduling, which divides iterations equally among threads
- Leverages multiple CPU cores to increase memory bandwidth utilization

3.2.4 CUDA GPU Implementation

The CUDA implementation offloads the operation to the GPU:

```
__global__ void stream(size_t nx, const double *__restrict__ src,
    double *__restrict__ dest) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < nx)
        dest[i] = src[i] + 1;
}
```

Listing 7: CUDA Stream Implementation

Key features:

- Each thread processes one array element independently
- Threads are organized into blocks, with multiple blocks forming a grid
- Includes bounds checking to handle edge cases when the number of threads exceeds the array size
- Requires data transfer between host and device memory

3.2.5 Memory Handling in CUDA Implementation

The CUDA version includes additional steps for memory management:

```
// Allocate device memory
double *d_src, *d_dest;
cudaMalloc(&d_src, nx * sizeof(double));
cudaMalloc(&d_dest, nx * sizeof(double));
cudaMemcpy(d_src, src, nx * sizeof(double), cudaMemcpyHostToDevice);

// ... [Kernel execution] ...

// Copy result back to host
cudaMemcpy(src, d_src, nx * sizeof(double), cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_src);
cudaFree(d_dest);
```

Listing 8: CUDA Memory Management

This demonstrates the typical CUDA memory workflow:

1. Allocate memory on the GPU
2. Copy input data from host to device
3. Execute the kernel
4. Copy results from device to host
5. Free GPU memory

3.3 Performance Measurement

The benchmark uses several metrics to evaluate performance:

1. **Elapsed Time:** The total time taken to perform all iterations
2. **MLUP/s:** Million Look-Ups Per Second, indicating computational throughput
3. **Bandwidth (GB/s):** The memory bandwidth achieved, calculated as:

$$\text{Bandwidth} = \frac{(\text{numReads} + \text{numWrites}) \times \text{sizeof(double)} \times \text{nCells} \times \text{nIt}}{\text{elapsedSeconds}} \tag{4}$$

3.4 Results and Analysis

Based on the data in `bandwidth_results.csv`, we can analyze how each implementation performs across different buffer sizes.

3.4.1 Bandwidth Performance

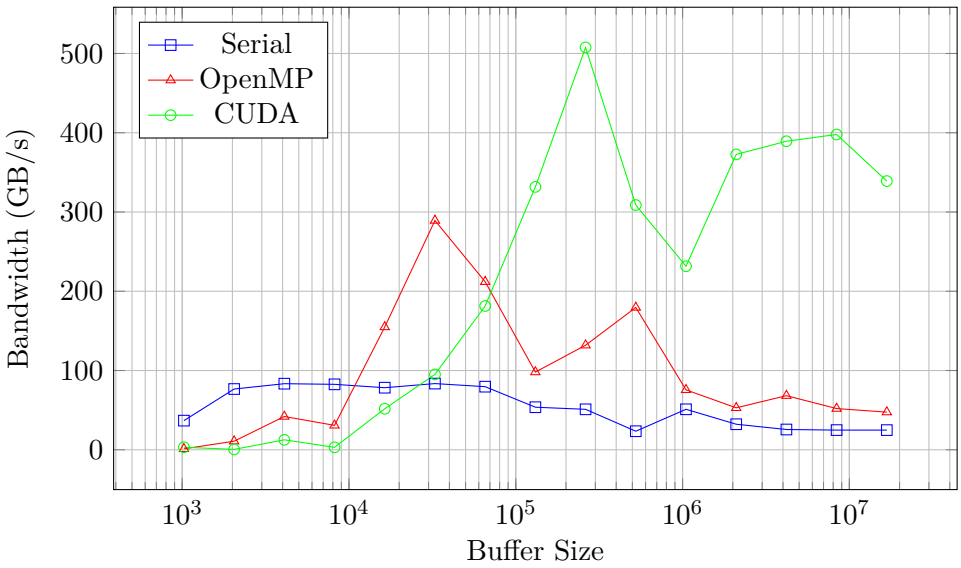

Figure 7: Bandwidth Performance Comparison

3.4.2 Key Observations

From analyzing the benchmark results:

1. **Serial CPU Performance:**
   - Performs surprisingly well for small buffer sizes (up to around 32KB)
   - Peak bandwidth of approximately 83.5 GB/s at 32KB buffer size
   - Performance drops significantly for large buffer sizes, stabilizing around 25 GB/s
   - This pattern suggests good cache utilization for small buffers but is limited by memory bandwidth for larger buffers
2. **OpenMP CPU Performance:**
   - Shows poor performance for very small buffers (overhead dominates)

- Peaks at around 289 GB/s with 32KB buffer size
- Shows significant variability across buffer sizes
- Generally outperforms the serial version for medium to large buffer sizes
- Performance pattern indicates the impact of parallel overhead, cache coherence, and memory bandwidth saturation

3. **CUDA GPU Performance:**
    - Poor performance for very small buffer sizes due to kernel launch and data transfer overhead
    - Demonstrates superior performance for large buffer sizes, reaching nearly 398 GB/s
    - Shows an upward trend until around 8MB buffer size, then starts to decline
    - Clearly dominates the CPU implementations for buffer sizes above 128KB
    - Performance pattern reflects the GPU's high memory bandwidth capabilities when properly utilized

3.5 Performance Crossover Points

The data reveals interesting crossover points where one implementation begins to outperform another:

1. **Serial vs. OpenMP:** OpenMP begins to consistently outperform serial at buffer sizes around 16KB
2. **OpenMP vs. CUDA:** CUDA becomes superior to OpenMP at buffer sizes around 128KB
3. **All Implementations:** For buffer sizes below 8KB, the serial implementation often delivers the best performance due to minimal overhead

These crossover points illustrate an important principle in parallel computing: the optimal implementation depends on the problem size, with different overheads dominating at different scales.

4 Technical Insights

4.1 Importance of Thread Organization

Both assignment components demonstrate the critical importance of effective thread organization in GPU programming:

- **Julia Set Generator:** Uses a 2D grid of 2D thread blocks (16×16 threads per block) to efficiently map threads to pixels
- **Stream Benchmark:** Uses a 1D grid of 1D thread blocks (256 threads per block) for processing array elements

These different approaches highlight how thread organization should match the underlying problem structure for optimal performance.

4.2 Memory Access Patterns

Both assignments also showcase the importance of efficient memory access patterns:

- **Julia Set Generator:** Each thread accesses image pixels in a coalesced manner, where adjacent threads access adjacent memory locations
- **Stream Benchmark:** The simple operation maximizes memory bandwidth utilization through straightforward access patterns

The performance results clearly demonstrate how memory access patterns significantly impact overall throughput, especially for memory-bound operations.

4.3 Kernel Launch Overhead

The Stream Benchmark results particularly highlight the impact of kernel launch overhead:

- For small buffer sizes, the CUDA implementation performs poorly due to the overhead of launching kernels and transferring data
- This overhead becomes negligible as the buffer size increases, allowing the GPU's superior memory bandwidth to become the dominant factor

This emphasizes the importance of amortizing kernel launch costs by processing substantial amounts of data per kernel invocation.

5 Conclusion

Key takeaways include:

1. GPUs excel at problems that can be decomposed into many independent tasks, as demonstrated by both the Julia set generation and stream operations
2. The optimal choice of parallel implementation (serial, OpenMP, or CUDA) depends significantly on the problem size
3. For memory-bound operations, GPUs offer substantial advantages for large data sets, but the overhead of data transfer and kernel launches can dominate for small problems
4. Effective thread organization and memory access patterns are critical for achieving peak performance in GPU programming