Project Report: Enhancing Image Processing Through Parallelization and Heterogeneous Computing

Team Members –

Abhishek Anand – [S20230010003]

Aku Rishita – [S20230010008]

Sai Teja – [S20230010011]

Introduction -

In this project, we focus on improvising the Openmp library using the CUDA Graph API to actively use the GPU [accelerators] to boost the performance by using Graph structure to improvise computation.

The Problem:

- Modern computing platforms are increasingly heterogeneous, often including GPUs for massive parallelism. But OpenMP implementations are unable to exploit graph paradigm available on GPUs
- While OpenMP is a standard for parallelising CPU applications and includes target constructs to offload work to accelerators, it doesn't always fully exploit the latest computing optimisations available on GPUs due to architectural differences.
- CUDA offer fine-grained control but are tedious and non-portable for programming heterogeneous applications, requiring explicit management of resources, communications, and synchronisations.

We are implementing and optimizing image processing operations to demonstrate parallel computing techniques, drawing inspiration from advanced heterogeneous computing concepts. Currently, we have developed a sequential C code for various image processing tasks using the STB image library. These operations include:

- **Grayscale Conversion**: Transforms a color image to grayscale by applying weighted averages to RGB channels for each pixel.

- **Gaussian Blur**: Applies a convolution filter to smooth the image, reducing noise and detail using a Gaussian kernel.
- **Sharpening Filter**: Enhances edges and details by applying a 3x3 convolution kernel that amplifies differences between pixels.
- **Noise Addition**: Introduces Gaussian noise to simulate real-world image degradation, using random number generation for each pixel.
- **Edge Detection**: Detects edges using the Sobel operator, involving dual convolutions for horizontal and vertical gradients followed by magnitude calculation.
- **Multi-level Image Compression**: Performs iterative Gaussian pre-filtering and downsampling to create compressed versions of the image at multiple levels.

Although Image processing is not directly related to our research paper. However, we use image processing as a demonstration vehicle to showcase how such optimizations could improve metrics like execution time and scalability. By profiling the sequential code, we identify hotspots and explore parallelization, setting the stage for integrating the paper's concepts in future.

Understanding the Research Paper and Its Relevance -

The paper addresses challenges in heterogeneous computing, where modern systems combine CPUs and GPUs for massive parallelism. OpenMP, a directive-based standard for CPU parallelization, uses target constructs to offload work to GPUs but often fails to exploit GPU-specific optimizations like graph execution due to architectural mismatches.

The authors propose enhancing OpenMP with the taskgraph directive, transforming task dependency graphs (TDGs) into device graphs (e.g. - CUDA Graph on NVIDIA GPUs, HIP Graph on AMD). Key contributions include:

- A proof-of-concept implementation in LLVM 17.0, converting OpenMP taskgraphs to CUDA Graphs, reducing overhead by skipping repeated task creation and dependency resolution.
- New clauses like target_graph for device graphs with host nodes and nowait for asynchronous execution.
- Efficient handling of dependencies, kernel nodes, host nodes, and memory movements (via map clauses) within the graph.

The CUDA Graph API works by recording operations (kernels, memory copies) into a graph during the first execution, then replaying it efficiently. It assumes task independence by default, aligning with OpenMP's task model, unlike the serial CUDA Stream API which requires extra events for concurrency. Taskgraph reduces contention, scales better on many-cores, and enables seamless interoperability with GPU APIs without code rewrites.

In our demo, we apply these ideas to image processing hotspots. While not the paper's domain, it allows us to measure improvements in a controlled, compute-intensive scenario, validating the library's potential for real-world applications.

Hotspot Identification and Parallelization Strategy -

To identify hotspots, we profiled the sequential code using tools like gprof on a system with an Intel i5 CPU and NVIDIA RTX GPU. We measured execution times for each function over multiple runs with large images (e.g., 4K resolution). Hotspots were determined by operations consuming >10% of total time.

These hotspots are parallelizable due to **data independence** (e.g., per-pixel operations) and **structured patterns** (e.g., convolutions). We chose a hybrid platform: OpenMP for CPU parallelism (simple directives, portability) and CUDA for GPU offloading (high throughput for image data).

Justification: OpenMP builds on our sequential C code with minimal changes, while CUDA aligns with the paper's GPU graph focus for massive parallelism. MPI was not selected as it's better for distributed systems, not single-node heterogeneous setups.

Future Plans: Scaling to Video Processing with CUDA Integration

For the next evaluation, we plan to extend to video processing, where videos are sequences of image frames. This is computationally expensive due to real-time requirements and large data volumes, making it an ideal test for heterogeneous computing. We will integrate proper CUDA Graph APIs with OpenMP taskgraph, as per the paper. Unlike basic OpenMP parallelism, CUDA will handle GPU-specific features like shared memory and streams, targeting speedups of 10x+ on NVIDIA hardware.

Identified Hotspots and Parallelization Rationale -

Below are the hotspots from profiling, with reasons for parallelizability:

- **HOTSPOT 1: Grayscale Conversion –**

convert_to_grayscale() function uses Independent pixel operations.

Parallelization Opportunity: Each pixel can be processed independently. OpenMP: Parallel for loops over pixels. CUDA: One thread per pixel.

Reason: No data dependencies; embarrassingly parallel.

- **HOTSPOT 2: Gaussian Blur Convolution**

apply_gaussian_blur() function uses 2D convolution with large kernels.

Parallelization Opportunity: Each output pixel computed independently. OpenMP: Parallel loops over output pixels. CUDA: Shared memory optimization for kernel coefficients.

Reason: Convolution is data-parallel; neighborhood accesses can be cached.

- **HOTSPOT 3: Sharpening Filter**

apply_sharpening_filter() function uses 3×3 convolution operation.

Parallelization Opportunity: Independent convolution per pixel. OpenMP: Parallel pixel processing. CUDA: Fast 3×3 convolution kernels.

Reason: Small kernel allows efficient parallel access without conflicts.

- **HOTSPOT 4: Gaussian Noise Addition**

add_gaussian_noise() function uses Random number generation + pixel operations.

Parallelization Opportunity: Parallel random number generation. OpenMP: Thread-local random states. CUDA: CUDA random number generators (cuRAND).

Reason: Independent per-pixel noise; thread-safe RNG avoids bottlenecks.

- **HOTSPOT 5: Edge Detection (Sobel)**

apply_edge_detection() function uses Dual convolution operations (X & Y gradients).

Parallelization Opportunity: Two parallel convolutions + magnitude calculation. OpenMP: Parallel loops with reduction operations. CUDA: Optimized gradient calculation kernels.

Reason: Gradients computable independently; magnitude is pointwise.

- **HOTSPOT 6: Gaussian Pre-filtering**

apply_gaussian_prefilter() function uses Same as Gaussian blur but repeated multiple times.

Parallelization Opportunity: Multiple blur operations in compression pipeline. OpenMP: Parallel convolution operations. CUDA: Streaming operations for pipeline processing.

Reason: Reusable blur kernel; pipeline stages are independent per level.

- **HOTSPOT 7: Image Downsampling**

downsample_image() function uses Block averaging and memory operations.

Parallelization Opportunity: Independent block processing. OpenMP: Parallel block processing. CUDA: Memory coalescing optimization + parallel averaging.

Reason: Blocks don't overlap; averages are local reductions.

- **HOTSPOT 8: Multi-level Compression Pipeline**

compress_image_multilevel() function uses Sequential pipeline of operations.

Parallelization Opportunity: Pipeline parallelism + data parallelism. OpenMP: Task-based parallelism for pipeline stages. CUDA: Graph execution for dependent operations (perfect for research!).

Reason: Levels have dependencies but internal ops are parallel; graphs handle sequencing efficiently.