**CUDA Image Processing Kernel - Technical Report**

**Objective**

The program implements a CUDA-based convolution kernel for processing a color image. The user specifies the desired filter by choosing a kernel from a predefined list. The kernel size is fixed at 3x3.

**Memory Allocation and Data Transfer**

**Memory Allocation**

1. **Host Memory:** The host allocates memory for both the input and output images.
2. **Device Memory:** CUDA memory allocation (cudaMallocAsync) is used to allocate memory for the input and output images on the device.
3. **Constant Memory:** The kernel weights, image dimensions, and channel information are stored in constant memory for fast access during kernel execution.

**Data Transfer**

* The host copies the input image data to the device memory using cudaMemcpyAsync.
* After processing, the device transfers the processed image data back to the host memory asynchronously.

**Stream-based Parallelization**

The program uses **4 CUDA streams** to process the image in parallel. The image is divided into horizontal segments, with each stream responsible for a specific number of rows. Overlapping regions (determined by the kernel radius) ensure accurate convolution at the boundaries.

**Stream Workflow**

1. Allocate memory for image chunks.
2. Transfer chunks from host to device asynchronously.
3. Launch the convolution kernel for each stream with corresponding image rows.
4. Transfer results from device to host asynchronously.
5. Synchronize all streams to ensure complete execution.

**Kernel Behavior**

The kernel performs the following operations:

1. Allocates a shared memory matrix of size w\_y \* w\_x to store a tile of the image, including neighbours pixels datas.
2. Loads data into shared memory in two phases to ensure full tile coverage. The first load will load and fill the upper part of the matrix; the second load will fill the lower one.
3. Each thread computes the convolution for a single pixel.
4. The final pixel value is clamped to the range [0, 255] and written to the output image array.
5. The process is repeated for every channel.

**Performance Metrics**

**Observations**

1. **Speedup:** The program achieves an **8.37x speedup** over the sequential version for an image size of 20,000x20,000 pixels.
2. **Kernel Execution Time:** The kernel itself processes the image in approximately **0.08ms**.
3. **CUDA Overheads:** Data transfer and the other CUDA overheads account for **1530ms** of the total execution time.
4. **Image building:** Image building timing is near **700ms**.

**Profiling Results**

* **Compute Throughput:** 67.78%
* **Memory Throughput:** 67.78%
* **L1 Throughput:** 98.27%
* **L1 Hit Rate:** 71.89%
* **L2 Throughput:** 61%
* **L2 Hit Rate:** 91.11%
* **Occupancy:** 94.23%
* **Active Warps per SM:** 30.15
* **Branch Efficiency:** 99.68%
* **Uncoalesced Access:** 83%

**Potential Improvements**

1. **Data Transfer Optimization:** Minimize the overhead caused by host-device memory transfers by optimizing the transfer size and frequency.
2. **Memory Access Coalescence:** Improve global memory access patterns to reduce uncoalesced memory accesses and enhance memory throughput.
3. **Image building:** As the building of the image takes around 700ms, this represents a room of improvement that can be achieved by parallelizing this step too.

**User-selectable Kernels**

The program supports the following 3x3 kernels:

1. **Identity:** No effect on the image.
2. **Blur:** Applies a Gaussian blur effect.
3. **Emboss:** Highlights edges with a directional light effect.
4. **Sharpen:** Enhances edge contrast.
5. **Outline:** Detects image edges.
6. **Bottom Sobel:** Detects vertical edges.
7. **Ridge:** Highlights ridges in the image.
8. **Edge Detection:** Detects edges using a Laplacian filter.
9. **Box Blur:** Averages the surrounding pixels.

**Conclusion**

The CUDA-based convolution kernel demonstrates performance improvements over the sequential approach, achieving good speedup even for large images. Profiling suggests further room for optimization, particularly in memory transfer and access patterns. By leveraging multiple streams and shared memory, the implementation efficiently handles the computational demands of image processing.