

GPU-accelerated Kernel Image Processing

Parallel Programming for Machine Learning course final project

Leonardo Bessi

E-mail address

leonardo.bessi@edu.unifi.it

Abstract

This project aims to compare a naive sequential algorithm for 2D image convolution kernels with a GPU-accelerated implementation. The parallel implementation was developed using the AMD ROCm platform and the HIP API. Using common GPU optimisation techniques, such as shared memory tiling and constant memory caching for filter coefficients, the GPU implementation achieved a significant speedup compared to the naive sequential baseline.

Future Distribution Permission

The author of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduction

In digital image processing, a kernel refers to a convolution matrix or mask that, applied to an image via a 2D convolution operation, enables effects such as blurring, sharpening, embossing, and edge detection.

The computation of each pixel depends only on its neighboring input pixels within the mask radius. The algorithm is "embarrassingly parallel" by offloading the computation to massively parallel processors, namely GPUs, the operation can be sped up significantly.

1.1. AMD ROCm and HIP API

The Heterogeneous-computing Interface for Portability (HIP) API, part of AMD's ROCm platform, is a C++ runtime API and kernel language ([1]). This project uses the HIP runtime and kernel language to implement the described GPU

kernels; this API is similar to NVIDIA CUDA, making it trivial to use for those familiar with the CUDA API. The portable nature of HIP means that HIP can target CPUs, AMD GPUs, and to some extent, even NVIDIA GPUs.

2. Implementation

The sequential implementation running on the CPU is straightforward and serves as the baseline for performance comparison.

This project brings four distinct GPU kernels, each introducing a specific optimization:

- **Naive:** this implementation is the simplest, as it maps blocks of threads directly onto image pixels, with each thread performing global memory accesses for both image data and mask coefficients.
- **Constant Mask Cache:** An improvement over the naive version that stores the convolution mask in the GPU's Constant Memory.
- **Tiled with Constant Cache:** This version introduces Shared Memory Tiling. By loading image tiles into shared memory, global memory bandwidth pressure is significantly reduced. This kernel works for single-channel images, or planar images.
- **Interleaved Tiled with Constant Cache:** An evolution of the tiled approach that natively handles interleaved (RGB) images, avoiding the need for external channel reordering.

2.1. Interleaved vs Planar images

Most common image formats store pixels as interleaved channel values (*RGBARGBA...*) while planar images organize data by channel (*RRR...GGG...BBB...*). This can be a problem when accessing pixels on the GPU as accessing one channel at a time will not coalesce reads. Reordering an interleaved image into a planar format can improve kernel efficiency, but the added overhead must be taken into consideration. This approach is best when multiple kernels are applied before the image is transformed back into interleaved. The third kernel in this project can process planar images by being invoked multiple times with different memory offsets.

2.2. Tiling Approach

In this implementation, thread blocks are sized to match the output tile (16×16). Since the convolution requires a "halo" of neighboring pixels (the mask radius), the input tile loaded into Shared Memory is larger than the thread block itself. Pixels are loaded collaboratively in batches. Once the loading is complete the output tile is computed. In the interleaved version, each thread iterates over the channels for its assigned pixel. Through testing it was determined that 16x16 tiles yield the best performance.

2.3. Memory Management and RAII

To simplify development and prevent memory leaks, a CPP wrapper class for image buffers was implemented. This class leverages the RAII idiom to manage both host and device memory. Under the hood, it uses the `stb_image` library ([2]) for loading and saving. The class handles dynamic memory allocation on the GPU and data copying ensuring synchronization between host and device. All resources are automatically released by the destructor.

3. Performance Analysis

3.1. Test Platform

The benchmarks were conducted on a system featuring an integrated AMD GPU. It is worth noting that in this APU configuration, the VRAM is shared with the system's LPDDR5x memory.

- **GPU:** AMD Radeon™890M (16 Compute Units / 1024 Stream Processors)
- **Memory:** 32GB LPDDR5x @ 8000 MHz
- **Platform:** ROCm version 7.1.1

3.2. Test Cases and Methodology

The performance of the optimized interleaved tiled kernel was evaluated against the sequential CPU baseline across various resolutions and filter types. The selected resolutions are:

- 1600×1067
- 1616×1080
- 1920×1080
- 2238×2446
- 6000×2908

Each one was processed using four different convolution masks:

- **Sharpen** (3×3)
- **EdgeDetect** (3×3)
- **Gaussian Blur** (5×5)
- **Gaussian Blur** (9×9)

Execution times were recorded for the CPU, the GPU kernel only (excluding memory transfers), and the total GPU execution time (including HtoD and DtoH copies).

4. Results

The performance data collected is shown in Table 1, which provides a detailed comparison of execution times for each individual filter. Figure 1 illustrates the correlation between the input image resolution (expressed in total pixels) and the corresponding execution time.

Table 1: Performance comparison between CPU and GPU implementations across different filters and resolutions. All times are expressed in milliseconds (ms).

Resolution	Filter	CPU Time	GPU (w/ Mem)	GPU (Kernel)	Speedup (K)	Speedup (T)
1600×1067	Sharpen	254.14	4.54	0.57	447.5×	56.0×
	EdgeDetect	286.88	2.89	0.48	596.4×	99.4×
	Gaussian 5×5	691.05	3.21	0.99	694.6×	215.1×
	Gaussian 9×9	2410.49	4.79	2.61	924.7×	503.6×
1616×1080	Sharpen	279.00	3.61	0.72	385.6×	77.2×
	EdgeDetect	292.71	2.39	0.71	409.6×	122.4×
	Gaussian 5×5	702.51	2.95	0.90	780.2×	238.2×
	Gaussian 9×9	2459.42	4.74	2.67	922.4×	518.8×
1920×1080	Sharpen	318.90	2.93	0.85	375.4×	109.0×
	EdgeDetect	324.78	2.77	0.78	414.8×	117.2×
	Gaussian 5×5	842.00	3.25	1.25	671.7×	259.1×
	Gaussian 9×9	2923.94	5.07	3.03	964.0×	577.1×
2238×2446	Sharpen	787.50	7.44	2.13	368.9×	105.8×
	EdgeDetect	903.10	6.61	1.51	596.7×	136.6×
	Gaussian 5×5	2214.45	8.41	3.20	692.1×	263.3×
	Gaussian 9×9	7738.04	13.11	7.93	975.3×	590.1×
6000×2908	Sharpen	2538.67	24.30	4.80	528.6×	104.5×
	EdgeDetect	2896.76	25.39	6.89	420.2×	114.1×
	Gaussian 5×5	6995.43	27.79	10.80	647.6×	251.7×
	Gaussian 9×9	24690.00	40.70	27.60	894.5×	606.6×

4.1. Filter Chain Throughput Analysis

To further evaluate the computational efficiency, a sequence of kernels (*Sharpen* → *Gaussian 5×5* → *Edge Detect* → *Gaussian 9×9*) was executed. The following results represent purely the computational time for both CPU and GPU, excluding memory transfers and initialization overhead.

This test highlights the raw processing power of the GPU when executing multiple operations in sequence. As shown in Table 2, the GPU maintains a massive lead, processing the 6K resolution chain in approximately 107 ms, where the CPU requires over 37 seconds.

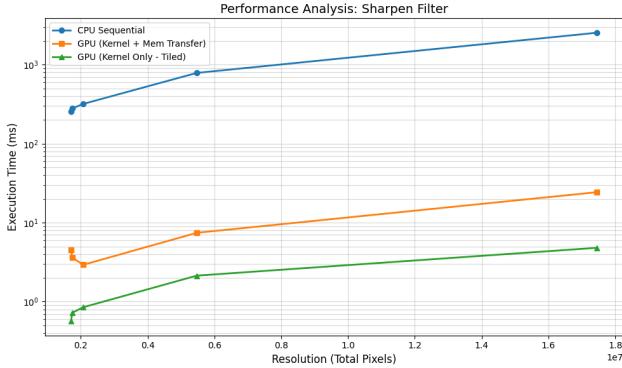
5. Conclusion

This project successfully demonstrated the performance advantages of GPU acceleration for spatial image filters. By implementing a tiled convolution kernel using AMD ROCm and HIP,

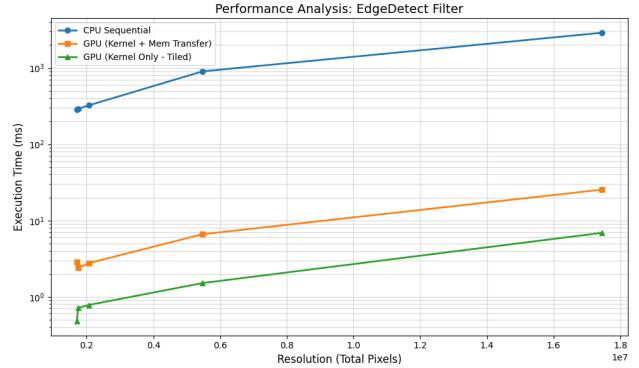
a peak speedup of over 600x on high-resolution images, was achieved.

References

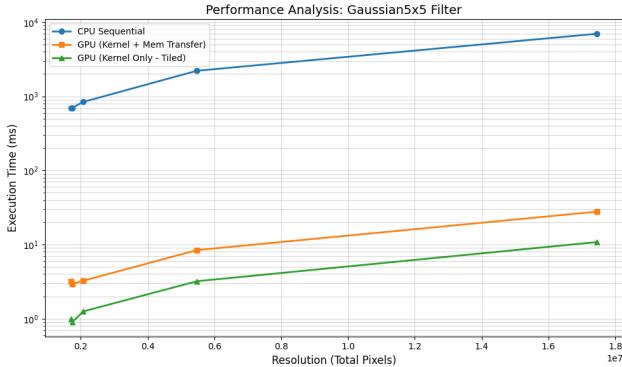
- [1] I. AMD, Advanced Micro Devices. Hip: Heterogeneous-computing interface for portability (rocm). https://rocm.docs.amd.com/projects/HIP/en/latest/what_is_hip.html, 2026. Accessed: 2026-01-21; ROCm HIP C++ runtime API and kernel language documentation.
- [2] S. Barrett. *nothings/stb*, 05 2021.



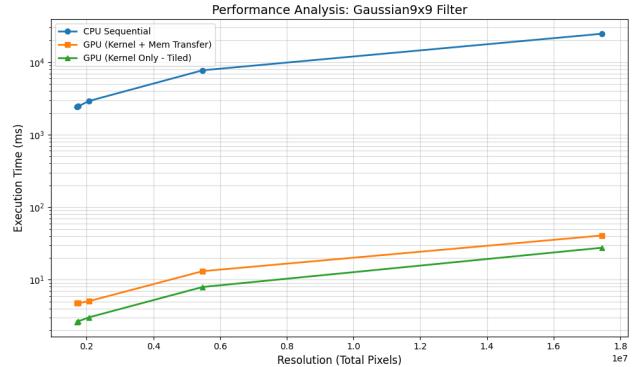
(a) Sharpen filter performance.



(b) Edge Detect filter performance.



(c) Gaussian Blur 5×5 performance.



(d) Gaussian Blur 9×9 performance.

Figure 1: Execution time analysis for different filters relative to image resolution (total pixels). The log-scale plots show the significant speedup achieved by the GPU implementation compared to the sequential CPU baseline.

Table 2: Filter Chain Computational Performance (Kernel-only time). Comparison of total processing time across the four-filter sequence.

Resolution	CPU Comp. Time (ms)	GPU Kernel Time (ms)	Speedup
1600×1067	3679.73	13.41	274.4×
1616×1080	3713.66	13.03	285.0×
1920×1080	4463.03	15.07	296.2×
2238×2446	11745.10	41.99	279.7×
6000×2908	37373.30	107.06	349.1×