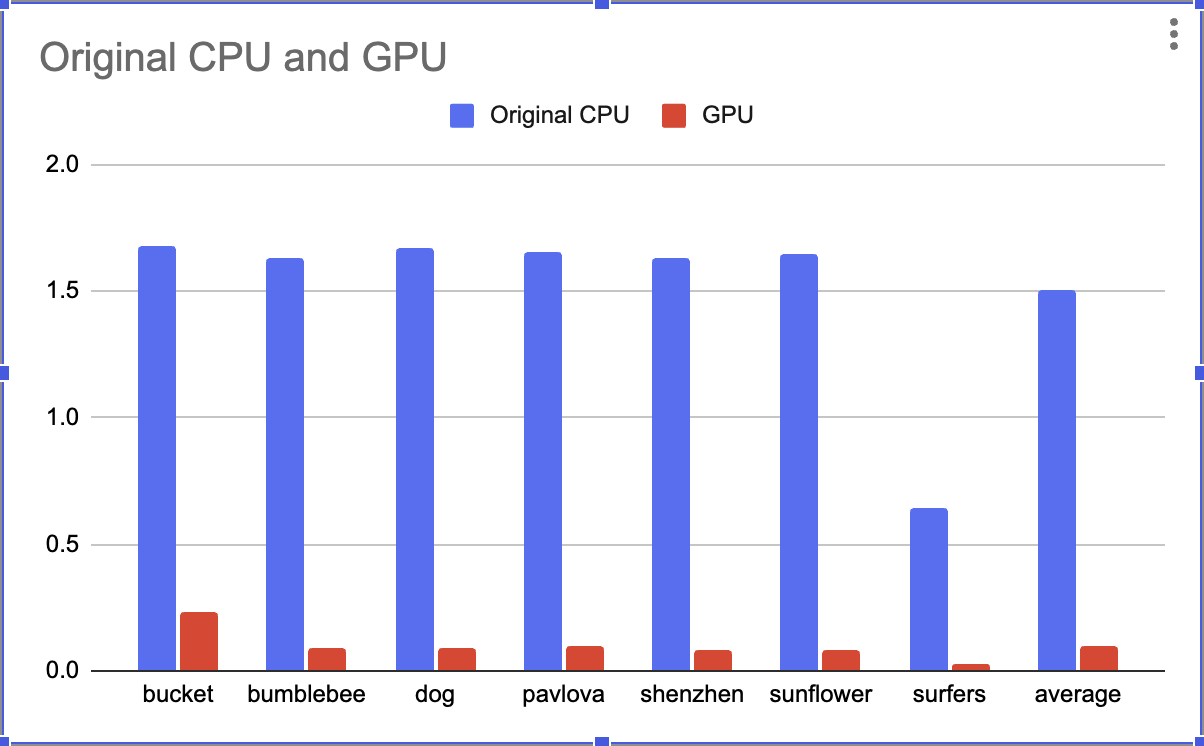
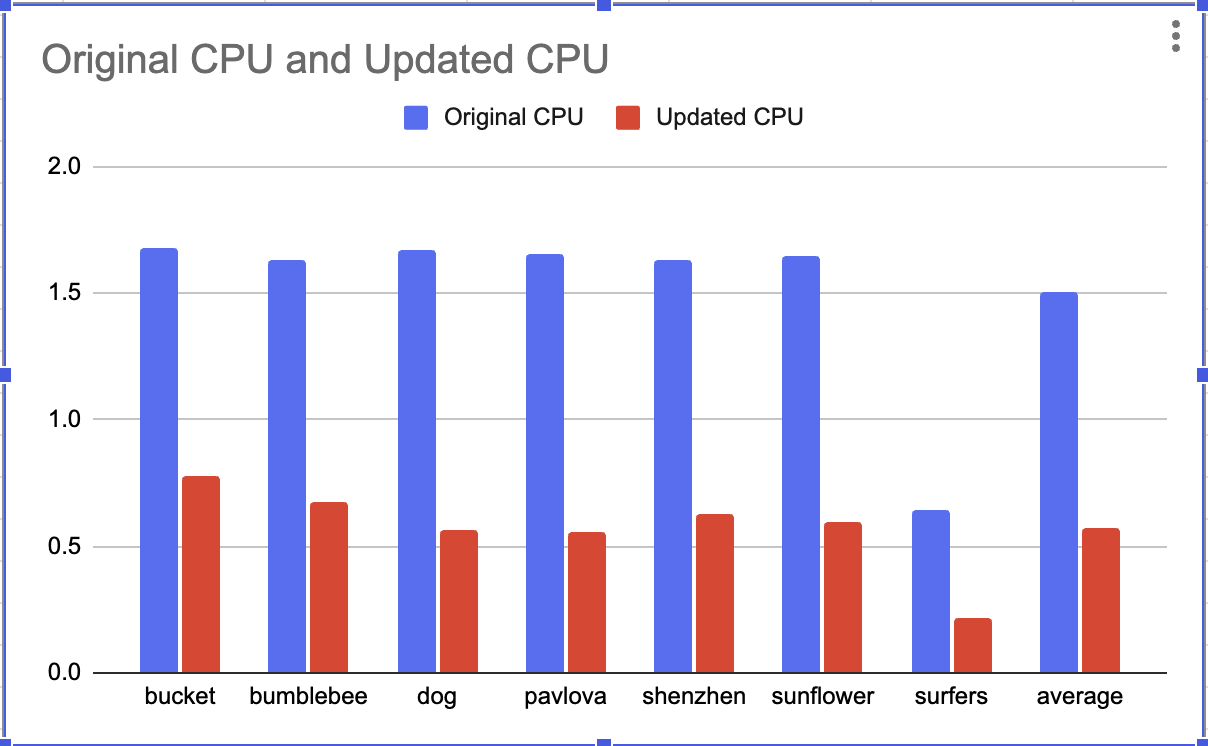
## Executive Summary

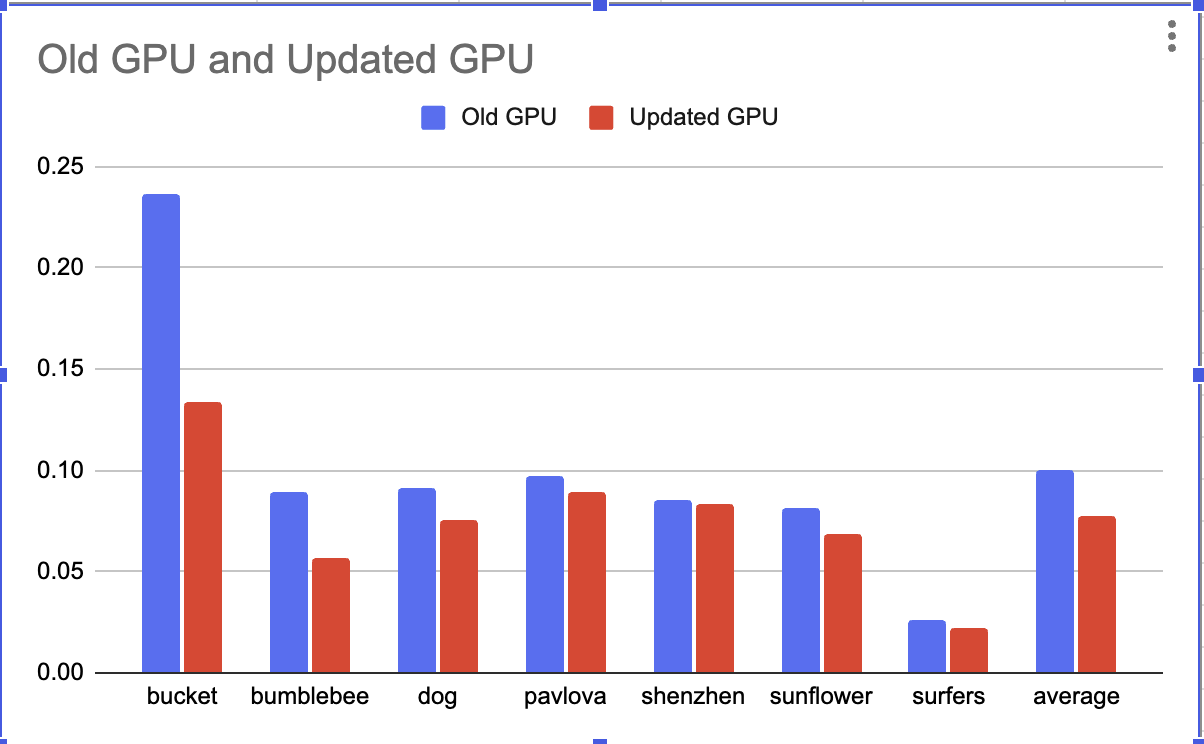
The CelAnim program has undergone substantial optimization on both CPU and GPU platforms. These optimizations have resulted in significant performance gains, enhancing the efficiency of the program.

### Key Findings

* **CPU Optimization**: Careful code refactoring, including reducing function calls within loops and eliminating redundant calculations, yielded a substantial performance increase. On average, the updated CPU code delivers a 62% speedup compared to the original implementation.
* **GPU Acceleration**: Leveraging the parallel processing capabilities of a GPU and optimizing code for constant memory usage led to notable improvements. The updated GPU implementation demonstrates an average speedup of 23% over the original GPU code.

### Performance Comparison Charts





### Optimized CPU Implementation

The primary goal of the CPU optimization was to streamline code execution within critical loops and reduce computational overhead.

* **Minimizing Nested Function Calls**: Function calls were identified within performance-sensitive loops and, where possible, their calculations were integrated directly into the loop to reduce function call overhead.
* **Precalculating Redundant Values**: Calculations that repeated the same result within loops were moved outside the loops and the result stored. This avoided unnecessary recalculations with each iteration.
* **Speedup**: These optimizations led to a substantial 62% average speedup across the test datasets.

### GPU Accelerated Implementation

This optimization leveraged the parallel processing power of a GPU to distribute the image processing workload across multiple cores.

* **Kernel Implementation**: The core image processing algorithm was translated into a GPU kernel, specifically designed to be executed by many threads simultaneously.
* **Memory Object sharing:** Same memory objects, **inputMem** and **outputMem**, are shared between different kernels to get better memory allocation.
* **Constant Memory Utilization**: Values that remained the same across multiple work-items, **filter values,** were stored in constant memory, which provides faster access for frequently used data.
* **Speedup**: The GPU implementation achieved an average speedup of 23% over the original GPU version.

### Difficulties During Implementation

The optimization process presented a few challenges as follow -

* Ensuring that the Java and OpenCL versions produced identical outputs. Even slight differences in how they handle data types could lead to inconsistencies, requiring careful attention.
* Pinpointing the exact causes of performance bottlenecks within the GPU's parallel architecture showed more complexity than on the CPU side.

### Future Improvements

For future improvements, focusing on fine-tuning the GPU kernel holds significant potential. Here are some key areas to explore:

* Enhanced Shared Memory Usage: Strategically using shared memory within workgroups can reduce the need for slower global memory accesses.
* Optimized Loop Structures: Experimenting with loop unrolling could improve instruction-level parallelism.
* Filter Parallelism: Investigate strategies to process different image filters in parallel, taking full advantage of the GPU's many cores.