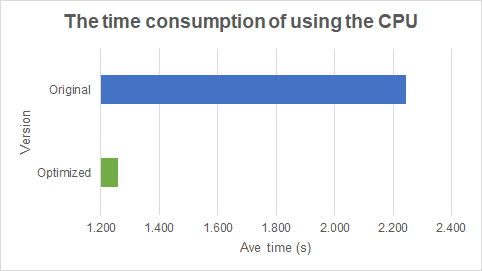
**Cartoon Images Processing Optimization Report**

**1. Executive Summary**

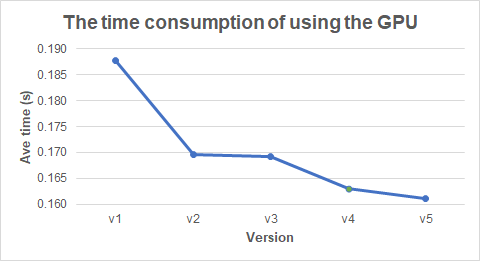
The primary goal of optimization is to enhance processing speed while maintaining output quality. There are two versions: CPU and GPU.

In the CPU version, after optimization, the average processing time for handling 7 images is reduced from 2.245s to 1.260s. The increase rate is 43.9%.

In the GPU version, after three versions’ optimization, the average processing time for handling 7 images is reduced from 0.188s to 0.161s. The increase rate is 16.5%.



***Figure 1*** *The average processing time for 7 images using the CPU*



***Figure 2*** *The average processing time for 7 images using the GPU*

**2. Optimization Approaches and Results**

**2.1 CPU version using Java**

**2.1.1 Optimization point 1**

Optimize *convolution()* method:

1. Cache *currentImage()* to reduce function calls

* Instead of calling *currentImage()* multiple times, store *pixels[currImage]* in a local variable image.
* This reduces overhead by avoiding repeated array lookups.

1. Cache *currentImage()* to reduce function calls

* Store the wrapped y-coordinates and x-coordinates in wrappedY and wrappedX.
* This eliminates repeated calls to *wrap()* inside the nested loops.
* *wrappedY[i]* includes *y \* width* computation upfront, reducing multiplications.

1. Inline *colourValue()* computation

Instead of calling *colourValue(rgb, colour)*, directly apply bit shifting and masking.

**2.1.2 Optimization point 2**

Optimize *wrap()* method:

* The optimized version uses branchless programming, which avoids unnecessary condition checks, leading to more efficient execution.
* Branching (i*f-else*) can disrupt this pipeline, whereas *Math.max()* and *Math.min()* are single-cycle operations that allow smooth execution. These two methods can be vectorized efficiently, improving overall throughput.

**2.2 GPU version using OpenCL**

**2.2.1 Version 1**

* Supplemented the kernel code based on the Java logic.
* Implemented memory object sharing in the *processPhotoOpenCL()* method.

**2.2.2 Version 2**

Optimize *sobelEdgeDetect()* and *mergeMask()* by using *select()* to remove conditional assignments:

* In GPU computing, conditional statements (*if* statements or the *? :* operator) can cause branching, which reduces parallel computing efficiency. This happens because different threads execute different paths, leading to thread synchronization overhead. The *select()* function can be optimized by the OpenCL compiler into bitwise operations or SIMD instructions, reducing the overhead of conditional jumps.
* Speedup: 9.69% improvement

**2.2.3 Version 3**

Use *#pragma unroll* in *convolution()* method and *sobelEdgeDetect()* method to enable loop unrolling:

* When the loop count is fixed and relatively small (5 and 9), using #pragma unroll can effectively reduce branching and loop control overhead.
* Speedup: 9.90% improvement

**2.2.4 Version 4**

Optimize the *processPhotoOpenCL()* method by setting up two command queues to parallelize the execution of *sobelKernel* and *reduceKernel*:

* The optimized code uses two command queues (queue1 and queue2) to execute different OpenCL kernels. *sobelEdgeDetect* and *reduceColours* run in parallel on queue2 to improve execution speed.
* Speedup: 13.21% improvement

**2.2.5 Version 5**

Optimized OpenCL pipeline with event-based synchronization：

* Replaced *clFinish()* with *cl\_event()* dependencies for better concurrency
* Improved memory management and event cleanup
* Speedup: 16.50% improvement

**3. Detailed Statistics**

* CPU: Intel® Core™ i5-10500 CPU @ 3.10GHz
* GPU: NVIDIA GeForce GTX 1050

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **The time consumption of using the GPU**  (Unit: second) | | | | | | | |
|  | **Run 1** | **Run 2** | **Run 3** | **Run 4** | **Run 5** | **Ave** | **Speedup** |
| **v1** | 0.217 | 0.167 | 0.174 | 0.212 | 0.169 | 0.188 | —— |
| **v2** | 0.182 | 0.164 | 0.170 | 0.157 | 0.175 | 0.170 | 9.69% |
| **v3** | 0.186 | 0.165 | 0.168 | 0.162 | 0.165 | 0.169 | 9.90% |
| **v4** | 0.163 | 0.164 | 0.166 | 0.163 | 0.159 | 0.163 | 13.21% |
| **v5** | 0.163 | 0.162 | 0.162 | 0.159 | 0.160 | **0.161** | **16.50%** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **The time consumption of using the CPU**  (Unit: second) | | | | | | | |
|  | **Run 1** | **Run 2** | **Run 3** | **Run 4** | **Run 5** | **Ave** | **Speedup** |
| **Original** | 2.254 | 2.231 | 2.241 | 2.251 | 2.250 | 2.245 | —— |
| **Optimized** | 1.263 | 1.256 | 1.265 | 1.258 | 1.256 | **1.260** | **43.90%** |

**4. Conclusion and Recommendations**

**4.1 Summary of Achievements**

* Utilized multiple command queues to execute different tasks in parallel, maximizing GPU resource utilization and reducing idle time.
* Shared memory objects between kernels to minimize unnecessary data transfers and improve memory efficiency.
* Applied loop unrolling in OpenCL kernels to reduce loop overhead and enhance computational performance.
* Replaced *if-else* conditions with the *select()* function to eliminate branch divergence and improve execution efficiency on the GPU.

**4.2 Faced Challenges**

* Debugging OpenCL code can be challenging when the image hash differs from the expected Java result. This discrepancy often arises from floating-point precision differences, parallel execution order variations, boundary handling inconsistencies, memory alignment issues, or unintended memory access.
* Debugging synchronization in multiple OpenCL command queues is challenging because tasks do not automatically execute in order across queues. Issues like race conditions, data inconsistency, and inefficient synchronization can arise.

**4.3 Suggestion for future improvements**

* Ensure the code is fully parallelized.
* To avoid bottlenecks, try to minimize memory transfer operations and use the appropriate memory types.
* When performing intensive image processing on the GPU, avoid frequently transferring data back and forth between the GPU and CPU unless necessary to retrieve results.
* Choosing the correct workgroup size.
* Use OpenCL performance analysis tools to identify bottlenecks and optimize performance.