System Programming HW2: Efficient Program Implementation

202111003 강소은

1. Implementation results

Image Size	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Img_128	3.5804	3.4362	3.4374	3.4344	3.5449	3.48666
Img_256	3.4638	3.4675	3.5570	3.6851	3.5074	3.53616
Img_512	3.5002	3.5069	3.5191	3.4844	3.6095	3.52402
Img_768	3.4798	3.4935	3.4779	3.4969	3.4730	3.48422
Img_1024	4.1481	4.1114	4.1553	4.1324	4.1222	4.13388

Table 1: Performance comparison between baseline and filter_optimized

Each image was tested for performance five times. The results indicated an approximate 3.51x improvement in performance for all images except Img_1024. For Img_1024, the performance enhancement was notably higher at 4.13x. The larger image size results in higher memory access frequency, increased loop iterations, and a greater chance of cache misses compared to smaller images. Therefore, the optimizations showed a more significant improvement for larger images. However, the performance did not scale proportionally with image size, which suggests the need for further analysis. Currently, the code seems to be optimized to achieve the highest performance for Img_1024, making it the benchmark for our optimizations.

2. Optimization approach:

The following optimization strategies were applied in sequence during the assignment:

1) Removal of Dynamic Memory Allocation

The original code dynamically allocated and deallocated memory for the Pixel structure to store Convolution results. In the optimized code, this overhead was eliminated by removing dynamic memory allocation.

2) Improved Cache Locality for Image Processing

The original code processed the image in a column-row order. The optimized code processes data row-wise to enhance cache locality. This optimization reduces cache misses by improving memory access patterns.

3) Elimination of Repeated Calculations

The original code repeatedly computed indices for input and filter. To reduce the frequency of these computations, repetitive results were stored in local variables outside the loop. However, this did not show a significant performance improvement or negligible performance drop for some indices. Given the negligible performance change and improved code readability, this strategy was selectively applied.

The combined improvements from steps 1), 2) and 3) resulted in approximately 1.7x performance enhancement.

4) Removal of Procedure Calls

In the original code, the filter_baseline() function called convolution() to compute the convolution for each pixel. The optimized version, filter_optimized(), performs the convolution directly within the function, eliminating the overhead of procedure calls.

5) Efficient Boundary Checking

Instead of executing boundary checks for all pixels, the optimized code separates internal and boundary pixels. Internal pixels bypass the boundary checks, resulting in increased computation speed.

Steps 4) and 5) together contributed to a performance gain of approximately 2.6x.

6) Improved Filter Element Access Time

Direct use of the filter variable necessitates memory access for each use. By storing filter elements in local variables, the optimized code reads the values from memory once and subsequently uses CPU registers, enhancing the computation speed.

7) Convolution Loop Unrolling

The loop unrolling technique was applied to the convolution operations to reduce iteration counts. Given the fixed 3x3 filter size, loop unrolling is feasible and minimizes redundant operations compared to a double for-loop approach. This reduces loop overhead and improves CPU pipeline efficiency. Additionally, data is accessed row-wise within the loop to maximize cache locality, and index computations use addition instead of multiplication for faster operations. Practical optimization of cache locality was achieved through an efficient computation order.

Steps 6) and 7) combined led to a performance improvement of approximately 4.1x.