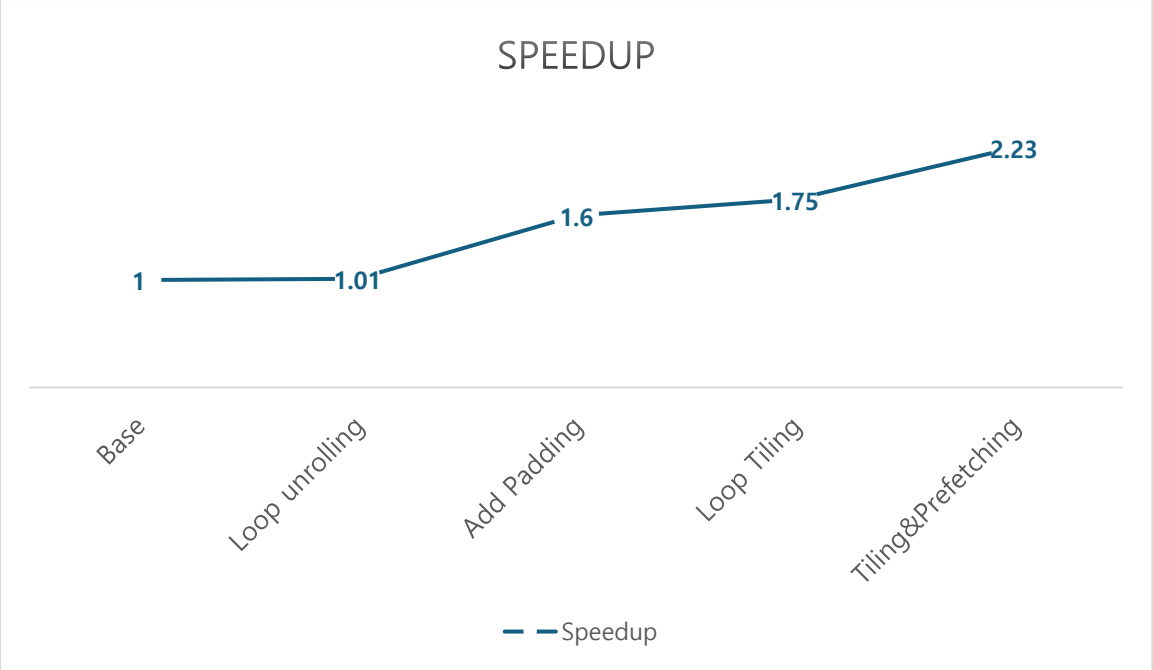
System_Programming_hw2_report_202011037_김민재

Section 1: Implementation Results

The Graph below presents the speedup achieved by the optimization approach implementation compared to the baseline(unoptimized) code for various optimization strategies. The values presented are the mean speedup ratios obtained from five trials for each configuration.



Section 2: Optimization Approach

	32	64	128
128	1.85	1.92	1.85
256	1.90	1.87	1.93
512	1.90	1.90	1.92
768	1.86	1.88	1.88
1024	2.11	2.12	2.23

The table above presents the speedup achieved by the optimized image filtering implementation compared to the Vaseline code for various image sizes and tile configurations. The values presented are the mean speedup ratios obtained from five trials for each configuration.

The results demonstrate that the optimized implementation consistently outperforms the baseline across all image sizes and tile configurations. The highest speedup for each image size was generally achieved with a tile size of 128, except for the 128-pixel image where the tile size of 64 performed slightly better. Notably, when the image size reached 1024 pixels, a significant jump in speedup exceeding 2x was observed, differentiating it from the performance gains seen with smaller image sizes.

The optimization strategy employed in this work encompasses three key techniques: fixed-point arithmetic, tiling, and prefetching.

Fixed-point arithmetic: Replacing floating-point calculations with faster integer operations.

Tiling: Dividing the image into smaller blocks for efficient cache utilization.

Prefetching: Anticipating and loading data into the cache before it's needed to reduce memory access delays.

Initially, I focused on varying the tile size to maximize the optimization speedup ratio. After conducting five trials for each combination of tile size (32, 64, 128) and image size, and calculating the average speedup, it became evident that a tile size of 128 generally yielded the most significant improvements.

This observation can be attributed to the relationship between tile size and cache size. If the tile size is too small, frequent memory accesses are required to fetch the next tile's data while processing the current tile, leading to increased cache misses. Through extensive experimentation, I was able to identify the optimal tile size for each image size, striking a balance between cache efficiency and computational overhead.

I tried other strategies before.

Loop Unrolling: While loop unrolling can reduce loop overhead, the marginal gains in this specific scenario were not significant enough to warrant inclusion in the final optimized implementation.

Adding Padding: Padding the image borders can simplify boundary handling, but the added memory overhead and potential impact on cache efficiency outweighed the benefits in our case.

Loop Tiling: Loop tiling was initially considered as a separate optimization but was later combined with the general tiling strategy, as it offered a more holistic approach to cache optimization.