HW2

202011083 박영진

1. Speedup Table

Image Size	Speedup (filter_optimized)
img_128	3.192963
img_256	3.116358
img_512	3.335092
img_768	3.174251
img_1024	3.794612

2. Optimization Approach

Basically, I used some macros to avoid the overhead of function calls.

Prefetching

- Prefetch data into the cache before it is needed to reduce memory access latency.
- Used **'__builtin_prefetch(ptr, 0, 3)'** to prefetch data. It reduces the time spent waiting for data to be loaded from memory, particularly beneficial for large images.

Memory Alignment

- Align memory access to 16-byte boundaries to improve cache efficiency.
- Used **'__attribute__((aligned(16)))'** to align the input image data to 16-byte boundaries.
- Copied the input image to an aligned memory array to improve cache performance.

Loop Unrolling

- Process multiple pixels in a single loop iteration to reduce loop overhead and increase parallelism.
- This reduces the number of loop control instructions and allows for better utilization of CPU resources.
- Processed 8 pixels in parallel within the inner loop.
- Manually unrolled the loop to handle multiple pixels in each iteration.

Eliminating Dynamic Memory Allocation

- Removed dynamic memory allocation by deleting *malloc()* and *free()* calls for each pixel inside the loop to reduce overhead.
- Allocated a static array **aligned_input** with 16-byte alignment to store the input image data which makes it faster.

Other Factors

- Pointer arithmetic is used to navigate through arrays more efficiently, reducing the overhead associated with array indexing.
- Precomputing and storing filter offsets to avoid recalculating them within the loop.
    - Calculated filter weights once and stored them in an array to avoid recomputing them within the inner loop.
    - Prefetched data and aligned memory accesses to ensure calculations are not repeated unnecessarily.