System Programming Personal HW2

202011178 이태양

Implementation result

| size | img_128 | img_256 | img_512 | img_768 | img_1024 |
|---|---|---|---|---|---|
| speed up | 1.9596 | 1.9571 | 2.0034 | 2.0534 | 2.3324 |
| | 1.9498 | 1.9875 | 2.0207 | 1.9941 | 2.3807 |
| | 1.9436 | 1.9891 | 1.9892 | 1.9887 | 2.2789 |
| | 1.9979 | 1.9903 | 1.9725 | 2.0097 | 2.3556 |
| | 1.9513 | 1.9813 | 2.0152 | 2.0118 | 2.4126 |

**This assignment focuses on optimizing the convolution function without using SIMD or multithreading. There are three main changes to the original code for optimization.**

First, removed dynamic memory allocation (malloc) within the for loop. Continuously accessing memory to allocate and deallocate during each iteration of the loop is inefficient and can degrade performance due to frequent memory access overhead and potential memory fragmentation issues.

Second, minimized the use of loops, such as for loops, by using loop unrolling. Using multiple loops increases the computational overhead since it involves not just repeating the block of code but also incrementing the index and comparing it against a condition in each iteration. This additional computation can negatively impact the performance. Therefore, we opted to unroll the loops, performing multiple iterations' worth of work in a single loop iteration to reduce overhead.

Third, since the data we are analyzing is an image, which does not change dynamically, it is easier to predict the next data access. Hence, we used the prefetching technique. Prefetching helps in reducing memory access latency by loading the data into the cache before it is actually accessed.

Lastly, we optimized the memory access pattern to enhance cache hit rates by ensuring that the memory blocks are accessed and processed sequentially. This change helps in making better use of the cache, reducing cache misses, and improving overall performance.

**Compare Optimization**

Initial / Dynamic Memory Allocation Removal

The first optimization involved removing dynamic memory allocation (malloc) within the for loop. Continuously accessing memory to allocate and deallocate during each iteration of the loop is inefficient and can degrade performance due to frequent memory access overhead and potential memory

fragmentation issues. By eliminating malloc, the performance increased from 1 to 1.45.

**Use Loop / Loop Unrolling**

The next step was to apply loop unrolling to minimize the use of loops. Using multiple loops increases computational overhead since it involves not just repeating the block of code but also incrementing the index and comparing it against a condition in each iteration. This additional computation can negatively impact performance. Therefore, by unrolling the loops, multiple iterations' worth of work were combined into a single iteration, improving performance. This optimization increased performance from 1.45 to 1.85.

**Prefetching Y / N**

Since the data being analyzed is an image, which does not change dynamically, it is easier to predict the next data access. Hence, the prefetching technique was used to reduce memory access latency. Prefetching loads data into the cache before it is actually accessed, minimizing memory access latency. This increased performance from 1.85 to 2.

**Increasing Loop Unrolling Factor**

Lastly, the loop unrolling factor was increased to 4, 8, and 16. Increasing the loop unrolling factor allows more work to be done in a single iteration, further enhancing performance. As a result, performance increased from 2 to 2.1, and then to 2.3.