2024 Spring, System Programming HW2

201811118 이 규

1. Implementation Results

Note that the speedup exhibited significant variation each time.

| Image Resolution | Speedup |
|---|---|
| 128x128 | x4.05 |
| 256x256 | x3.98 |
| 512x512 | x3.93 |
| 768x768 | x3.86 |
| 1024x1024 | x3.60 |

2. Optimization Approaches

In this section, first, I discuss the methods applied for optimization and their impacts. Second, I introduce the methods that were intended to be implemented but resulted in performance degradation. Note that the speedup discussed in this section applies to processing a 1024x1024 image.

2.1. Applied Approaches

2.1.1. Removing Function Call

For the first, I remove the function call that calls “convolution” function in “filter_optimized” function. This is because additional instructions are required to pass arguments, return values, and change the program counter during the process of calling a function. Additionally, malloc and free operations for Pixel struct are no longer needed. While adding the inline keyword can provide a hint to the compiler, it's not always certain whether the function will actually be inlined. Therefore, instead of adding the inline keyword to the “convolution” function, I directly moved the code into “filter_optimized” function.

This approach shows a 1.1 times speedup compared to the original source code.

2.1.2 Removing the Inner Loop (originally in the convolution function) and Adding Zero Padding Explicitly.

The original convolution function contains a nested for-loop that iterates a total of 9 times. Using for loops involves repetitive addition and comparison operations on loop variables, which can be inefficient from an optimization perspective. Therefore, instead of using for loops, I directly perform calculations by indexing. For the same reason, we explicitly performed zero padding. In the original code, conditions like ((y + dy) < 0 || (y + dy) >= height) were checked for every pixel position, which is inefficient. Therefore, I explicitly added zero pixels around the boundary of the original (width, height) image to create an enlarged image of size (width+2, height+2).

Applying this approach additionally in section 2.1.1 resulted in a 1.7 times speedup compared to the original source code.

2.1.3 Using Appropriate Data Type.

The pixels of the image range from 0 to 255 as integer values, and considering the convolution operation, they only need to cover a range up to 255 * 9. Therefore, I changed the variables r, g, b representing

this to integer type instead of double type. Additionally, I converted the filter values from float to int, assuming that integer multiplication is faster than float multiplication. To ensure accuracy in calculations, I multiplied the filter by a specific value and then divided by the same value in the result. I chose the specific value considering the filters used. For the boxblur filter, I used 1/9, and for the gaussian filter, I used 1/16. Therefore, I selected the specific value as 144, which is the least common multiple (LCM) of 9 and 16.

Applying this approach additionally in section 2.1.2 resulted in a 2.6 times speedup compared to the original source code.

#### 2.1.4 Using Loop Unrolling.

Now, my code consists of two nested double for loops. The first is for zero padding, and the second is for the convolution operation. To reduce the additional instructions caused by for loops, loop unrolling can be utilized. I performed loop unrolling to varying extents for padding and convolution operations, and the highest performance was achieved when increasing the loop variables by 8 for padding and by 4 for convolution.

Applying this approach additionally in section 2.1.3 resulted in a 3.2 times speedup compared to the original source code.

#### 2.1.5. Adding Hints to the Compiler for Register Allocation.

As mentioned earlier, my code consists of two nested double for loops, with repetitive indexing performed in the innermost loop. In the case of padding, all indexing is based on two indices, and in the case of convolution, it is based on three indices. I anticipated that allocating these five values to registers would yield the highest performance, so I added the 'register' keyword.

Applying this approach additionally in section 2.1.4 resulted in a 3.6 times speedup compared to the original source code.

### 2.2. Not Applied Approaches.

#### 2.2.1. Saving Frequently Used Indices.

In the case of padding, the same index value is used 3 times in a single loop iteration, and in convolution, the same index value can be used up to 9 times. Each index always computes an integer value by adding it to the value designated as a register in section 2.1.5. To avoid this, I precomputed and stored calculations for all indices, enabling later implementation to solely involve indexing operations.

However, this led to a performance degradation. I considered two reasons for this. Firstly, the addition operation for integers itself is not burdensome. Implementing in this manner reduced the calculation of indices from 3 times per iteration in padding to just once. Secondly, the approach of storing and using many variables may have significantly increased the cache miss rate.

#### 2.2.2. Using Bit Operation.

I attempted to apply bit operations such as shift, and, not operations to constrain the range of r, g, and b values to 0-255.

However, this also led to a performance degradation. I think the reason is that using bit operations increased the computational workload compared to the original compare operations.