Exercise Set 4

Darina Öö

12.02.2024

1 Exercise 1

1.1 Methodology and Execution

We can optimize the nested loop by recognizing that division is more computationally expensive than multiplication and by precomputing the reciprocal of the c[i] values outside of the inner loop.The optimized code can be seen in the files optimized_A.cpp and optimized_B.cpp

```
g++ -O0 -o original_a_O0 original_A.cpp
./original_a_O0

g++ -O0 -o original_a_O3 original_A.cpp
./original_a_O3

g++ -O0 -o optimized_a_O0 optimized_A.cpp
./optimized_a_O0

g++ -O3 -o optimized_a_O0 optimized_A.cpp
./optimized_a_O3

//similarly for B
```

1.2 Original Loop A

The original loop A performed conditional arithmetic operations based on the index value.

The CPU times measured were as follows:

- With -O0 optimization: 0.057142 seconds
- With -O3 optimization: 0.008051 seconds

1.3 Optimized Loop A

The optimized loop A removed the conditional check for the majority of iterations by splitting the loop into two separate loops. The CPU times measured were as follows:

- With -O0 optimization: 0.045885 seconds
- With -O3 optimization: 0.004081 seconds

1.4 Discussion

The optimized loop A showed an improvement in CPU time for both -O0 and -O3 options compared to the original loop. This demonstrates the effectiveness of loop unrolling and minimizing conditional checks within the loop.

1.5 Original Loop B

The original loop B performed division operations within a nested loop. The CPU times measured were as follows:

- With -O0 optimization: 0.057958 seconds
- With -O3 optimization: 0.011421 seconds

1.6 Optimized Loop B

The optimized loop B precomputed the reciprocal of the divisor outside the inner loop, converting division operations into multiplication operations. The CPU times measured were as follows:

- With -O0 optimization: 0.057496 seconds
- With -O3 optimization: 0.004944 seconds

1.7 Discussion

The optimized loop B showed a significant performance improvement when compiled with -O3 optimization, likely due to the more efficient use of CPU resources by converting divisions to multiplications, which are computationally less expensive.

1.8 Conclusion

The optimization strategies employed in loops A and B resulted in significant performance gains when compiled with -O3 optimization level. This highlights the importance of considering both algorithmic optimizations and compiler optimization flags in performance-critical applications. It also underscores the compiler's ability to further optimize already efficient code

2 Exercise 2

The goal of matrix_product.cpp function is to to calculate various matrix products using do/for loops, without using any library or intrinsic functions. The program calculates the products $C = AB$, $C = A^T B$, and $C = AB^T$, where A^T represents the transpose of matrix A. The CPU time consumed for each product is compared between different matrix operations and optimization levels.

2.1 Compilation

Compile the C++ program using the following commands:

```
g++ -O0 -o matrix_product_O0 matrix_product.cpp
g++ -O3 -o matrix_product_O3 matrix_product.cpp
```

2.2 Step 2: Execution

Execute the compiled program using the following command:

```
./matrix_product_O0
./matrix_product_O3
```

Results The program prints the CPU time consumed for each matrix product calculation. Below are the CPU times (in seconds) obtained:

Matrix Product	CPU Time Used in O0 (seconds)
$C = AB$	5.80552
$C = A^T B$	6.70514
$C = AB^T$	5.77559

Matrix Product	CPU Time Used in O3 (seconds)
$C = AB$	2.01364
$C = A^T B$	2.022
$C = AB^T$	2.00869

2.3 Conclusion

The optimization level -O3 significantly improved the performance of the matrix product calculations compared to optimization level -O0. The CPU times for all three matrix products were reduced by more than half when compiled with optimization level -O3. Additionally, it's observed that the CPU times for different matrix products were relatively consistent across both optimization levels, indicating that the computational complexity of each operation remained relatively constant regardless of optimization level. However, the overall execution time was significantly reduced with optimization level -O3, showcasing the effectiveness of compiler optimizations in optimizing overall program performance.

3 Exercise 3

3.1 Generate C Files

Run the `generate_files.py` script to generate C files for different unrolling factors. This script generates source code files named `unrolled_loop_`n`.c`, where n represents the unrolling factor. Each file contains a loop unrolling implementation for the specified factor.

3.2 Compilation

Compile the generated C files using the following commands:

```
gcc unrolled_loop_1.c -o unrolled_loop_1_O0 -O0
gcc unrolled_loop_1.c -o unrolled_loop_1_O3 -O3
gcc unrolled_loop_2.c -o unrolled_loop_2_O0 -O0
gcc unrolled_loop_2.c -o unrolled_loop_2_O3 -O3
gcc unrolled_loop_4.c -o unrolled_loop_4_O0 -O0
gcc unrolled_loop_4.c -o unrolled_loop_4_O3 -O3
gcc unrolled_loop_8.c -o unrolled_loop_8_O0 -O0
gcc unrolled_loop_8.c -o unrolled_loop_8_O3 -O3
gcc unrolled_loop_16.c -o unrolled_loop_16_O0 -O0
gcc unrolled_loop_16.c -o unrolled_loop_16_O3 -O3
gcc unrolled_loop_32.c -o unrolled_loop_32_O0 -O0
gcc unrolled_loop_32.c -o unrolled_loop_32_O3 -O3
```

3.3 Execution

Execute the compiled programs using the following commands:

```
./unrolled_loop_1_O0
./unrolled_loop_1_O3
./unrolled_loop_2_O0
./unrolled_loop_2_O3
./unrolled_loop_4_O0
./unrolled_loop_4_O3
./unrolled_loop_8_O0
./unrolled_loop_8_O3
./unrolled_loop_16_O0
./unrolled_loop_16_O3
./unrolled_loop_32_O0
./unrolled_loop_32_O3
```

3.4 Results

The table below shows the CPU time used (in seconds) for different unrolling factors and optimization levels:

Unrolling Factor	Optimization Level	CPU Time Used (seconds)
1	-O0	0.006820
1	-O3	0.000001
2	-O0	0.004176
2	-O3	0.000002
4	-O0	0.004038
4	-O3	0.000002
8	-O0	0.003543
8	-O3	0.000002
16	-O0	0.003149
16	-O3	0.000002
32	-O0	0.002634
32	-O3	0.000002

- For all unrolling factors, the CPU time used decreases significantly when optimization level -O3 is applied compared to -O0.
- Higher unrolling factors generally result in lower CPU time used, indicating improved performance due to loop unrolling.
- Optimization level -O3 leads to minimal CPU time used for all unrolling factors, demonstrating the effectiveness of compiler optimizations.

3.5 Conclusion

Loop unrolling and compiler optimizations play crucial roles in improving program performance. By increasing the unrolling factor and applying optimization level -O3, the CPU time used can be significantly reduced, resulting in faster execution times.

4 Exercise 4

4.1 Method

The program ‘simd_test.cpp‘ performs a series of vector multiplication operations. It was compiled and executed with and without SIMD extensions using the ‘-ftree-vectorize‘ and ‘-fno-tree-vectorize‘ compiler flags, respectively, with the general optimization option ‘-O2‘.

5 Compilation and Execution

The program is compiled with the following commands:

```
g++ -O2 -ftree-vectorize -o simd_test_enabled simd_test.cpp
```

```
g++ -O2 -fno-tree-vectorize -o simd_test_disabled simd_test.cpp
./simd_test_enabled
./simd_test_disabled
```

It is then executed multiple times to measure CPU time for vector multiplications. The vector size is varied, I chose to do it for values 32, 128 and 2560 to observe the effect of SIMD at different data sizes.

5.1 Results

The output was exported into the output_ex4.txt file.

5.2 Vector Size 2560

With SIMD extensions **enabled**, the average CPU time was approximately 6.97897 seconds with a standard deviation of 0.258381 seconds.

With SIMD extensions **disabled**, the average CPU time was significantly higher at 26.657 seconds with a standard deviation of 0.182187 seconds.

5.3 Vector Size 128

With SIMD extensions **enabled**, the average CPU time was 0.352309 seconds with a standard deviation of 0.00221706 seconds.

With SIMD extensions **disabled**, the average CPU time increased to 1.70073 seconds with a standard deviation of 0.164869 seconds.

5.4 Vector Size 32

With SIMD extensions **enabled**, the average CPU time was 0.104048 seconds with a standard deviation of 0.00341571 seconds.

With SIMD extensions **disabled**, the average CPU time was 0.369644 seconds with a standard deviation of 0.0385832 seconds.

5.5 Conclusion

The results demonstrate a clear performance improvement when using SIMD extensions. For larger vectors (size 2560), SIMD-enabled runs were approximately four times faster than SIMD-disabled runs. For smaller vectors, the performance gain was less pronounced but still significant, with SIMD-enabled runs being over three times faster.