Rijish Ganguly

rg239

ECE 565

1. Unmodified source code:
   1. Optimization – O3

Output sum: 69999988

Runtime is in milliseconds

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Shortest |
| 10,000,000 | 20.26 | 20.13 | 20.47 | 19.80 | 24.19 | 20.97 | 19.80 |
| 100,000,000 | 207.12 | 200.54 | 199.86 | 200.88 | 201.48 | 201.97 | 199.86 |

* 1. Optimization – O2

Output sum: 69999988

Runtime is in milliseconds

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Shortest |
| 10,000,000 | 19.61 | 20.45 | 20.37 | 23.27 | 20.20 | 20.78 | 19.61 |
| 100,000,000 | 199.00 | 198.55 | 198.77 | 210.63 | 204.78 | 202.34 | 198.55 |

1. Processor Architecture – x86 (64-bits)

CPU Frequency: 1.8 GHz

OS type: macOS Mojave

Standalone system

* 1. Optimization I

Loop Fusion

Resulting Code:

Optimization – O3

Output sum: 69999988

Runtime is in milliseconds

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Shortest |
| 10,000,000 | 23.66 | 24.31 | 24.18 | 24.53 | 26.08 | 24.55 | 23.66 |
| 100,000,000 | 586.62 | 268.78 | 250.74 | 253.58 | 250.05 | 321.95 | 250.05 |

Optimization – O2

Output sum: 69999988

Runtime is in milliseconds

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Shortest |
| 10,000,000 | 32.75 | 43.28 | 33.38 | 25.91 | 39.06 | 34.876 | 25.91 |
| 100,000,000 | 260.04 | 255.84 | 255.38 | 249.36 | 253.94 | 254.91 | 249.36 |

The optimization didn’t improve the performance of our code. The performance worsened compared to the original unmodified source code. Loop fusion may not improve performance as two separate loops might perform better on an architecture because of data locality.

* 1. Optimization II

Loop Peeling

Resulting Code:

Optimization – O3

Output sum: 69999988

Runtime is in milliseconds

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Shortest |
| 10,000,000 | 28.28 | 29.20 | 21.35 | 21.36 | 27.41 | 25.52 | 21.35 |
| 100,000,000 | 242.08 | 202.04 | 201.59 | 201.11 | 201.97 | 209.75 | 201.11 |

Optimization – O2

Output sum: 69999988

Runtime is in milliseconds

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Shortest |
| 10,000,000 | 21.10 | 29.86 | 21.10 | 20.90 | 22.13 | 23.01 | 20.90 |
| 100,000,000 | 223.03 | 201.55 | 200.97 | 202.64 | 200.11 | 205.66 | 200.11 |

The optimization didn’t improve the performance of our code. The performance of the modified code is comparable to the unmodified source code. However, the performance of the unmodified source code is better by a few milliseconds.

1. Loop Transformations
2. Loop Fusion

Original Dependencies:

S1 => T S2 (loop independent)

S1 => O S3 (loop carried)

S2 => A S3 (loop carried)

After Loop Fusion, the dependencies become as follow:

S1 => T S2(loop independent)

S1 => O S3(loop carried)

S2 => T S3(loop carried)

Loop fusion is safe if and only if no data dependence between the nests becomes loop-carried data dependence of a different type. We observe that the condition is violated for the above loop fusion.

1. Loop Interchange

Original Dependencies:

S1 => A S1 (loop dependent)

We observe that there is a data dependence carried by the outer loop executed for i and j to another statement instance i’ and j’ where i < i’ and j > j’.

Example: (1,3) => (2,2)

Loop Interchange is safe if outermost loop does not carry any data dependence from one statement instance executed for i and j to another statement instance executed for i’ and j’ where (i < i’ and j > j’) OR (i > i’ and j < j’). Since this condition is violated, loop interchange won’t be safe

1. Loop Fission

Original Dependencies:  
S1 => T S2(loop dependent)

After loop fission:

S1 => T S2(loop dependent)

Loop Fission is safe if and only if statements involved in a cycle of loop-carried data dependences remain in the same loop and if there exists a data dependence between two statements placed in different loops, the dependence type must not change. There is no violation of condition for this case. Hence, it is safe to use loop fission.

1. Loop Transformation

Initial Code:

A screenshot of a cell phone

Description automatically generated

1. Loop Invariant

Pulling non-loop-dependent calculations out of the loop

A screenshot of a cell phone

Description automatically generated

1. Loop un-switching

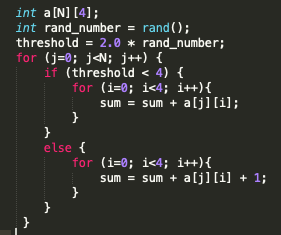
Move a conditional expression outside of a loop, and replicate loop body inside of each conditional block

A screenshot of a cell phone

Description automatically generated

1. Loop Interchange

Switch the positions of one loop that is tightly nested within another loop



1. Loop Peeling

Remove first and/or last iterations of a loop body to separate code outside the loop

A screenshot of a video game

Description automatically generated

1. Loop Unrolling

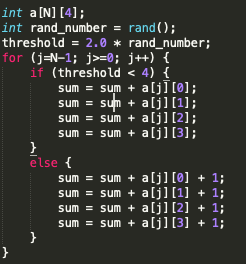
Combine multiple instances of the loop body and make corresponding reduction to the loop iteration count

A screenshot of a computer

Description automatically generated

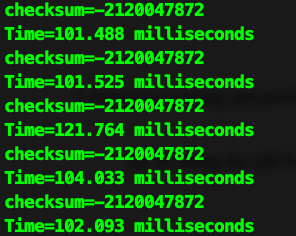
1. Loop Reversal

Reverse the order of the loop iteration



3. Function in-lining and performance

1. In-lining the add function



Average Time: 106.1806 ms

Shortest Run-time: 101.488 ms

Never In-lining the add function

A screen shot of a computer

Description automatically generated

Average Time: 208.7586

Shortest Run-time: 202.362 ms

1. Effects of in-lining

Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

Code-snippets for in-lining:

A screenshot of a cell phone

Description automatically generated

A black sign with white text

Description automatically generated

The loop being carried out

A close up of a sign

Description automatically generated

We don’t observe any explicit call to the addition function (\_Z3addii) as it gets expanded within another function

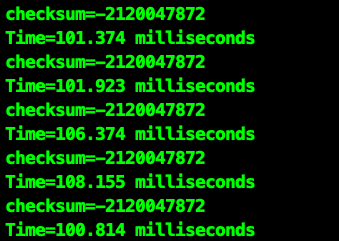
Code snippet-for no in-lining:

A picture containing black, wall

Description automatically generated

We observe an explicit call to the addition function

1. My measured performance results match my expectations. The average runtime of the program with always in-lining attribute was almost approximately half of the average runtime of the program with no in-lining. In-lining reduces the number of executed instructions by avoiding function calls and return instructions whereas context-switch can prove to be costly for small functions. For our program we are calling the addition function several times and since the size of the function is relatively small, we get superior performance with in-lining.
2. Performance of original code:



Average Run-time: 103.728 ms

Shortest Run-time: 101.374 ms

The performance of the original code is almost the same as the performance of the in-lined code. Based on this, we can make the claim that the compiler is in-lining the add function.

2.

(a)

**Iteration-space Traversal Graph**

A picture containing building

Description automatically generated

Iteration: (1,1) => (2,1) => (2,2) => (3,1) => (3,2)

(b)

Dependence:

1. S1[i,j] => T S3[i,j] Loop Independent
2. S1[i,j] => A S2[i,j] Loop Independent
3. S3[i+1,j] => T S1[i+2,j] Loop Carried
4. S4[i+1,j+1] => T S4[i+2, j] Loop Carried
5. S1[i,j] => T S2[i+1,j+1] Loop Carried
6. S1[i,j] => A S1[i+1,j-1] Loop Carried
7. S3[i,j] => T S2[i+2,j] Loop Carried

(c)

`