copt Writeup

Jackson Mowry

Fri Dec 5 18:07:21 2025

1 Optimization Details

1.1 Matrix Initialization

For the matrix initialization subroutine I found that eliminating function calls to helper functions (i.e. `check` and `set`) provided a significant speed-up.

Originally my plan was to use the `register` keyword for all local variables. Utilizing `register` tells the compiler to store our values directly in registers, avoiding writing them to memory when possible. When using `-O0` we're inhibiting the compiler from applying these register optimizations automatically, therefore adding it explicitly re-enables the compiler to generate more efficient assembly.

Additionally, I found a small speed up from computing the index of the matrix row and final column before accessing each matrix, helping to avoid duplicate computation of the final index. Without this "memoization" the compiler generates duplicate instructions when generating the address for `mat1` and `mat2`.

```
.L4:
        // mat1
        mov     eax, ebx                // Duplicate 1
        imul    eax, r12d               // Duplicate 2
        add     eax, r13d               // Duplicate 3
        cdqe                            // Duplicate 4
        sal     rax, 2                  // Duplicate 5
        add     rax, r14
        mov     DWORD PTR [rax], ebx
        // mat 2
        mov     eax, ebx                // Duplicate 1
        imul    eax, r12d               // Duplicate 2
        add     eax, r13d               // Duplicate 3
        cdqe                            // Duplicate 4
        sal     rax, 2                  // Duplicate 5
        lea     rdx, [r15+rax]
        lea     eax, [rbx+1]
        mov     DWORD PTR [rdx], eax
```

With “memoization” we get the following assembly.

```
.L4:
        // Memoized index computation
        mov     eax, edi
        lea     r14d, [rax+r13]
        // mat1
        movsx   rax, r14d               // Duplicate 1
        sal     rax, 2                  // Duplicate 2
        add     rax, r15
        mov     DWORD PTR [rax], ebx
        // mat2
        movsx   rax, r14d               // Duplicate 1
        sal     rax, 2                  // Duplicate 2
        mov     rcx, rsi
        lea     rdx, [rcx+rax]
        lea     eax, [rbx+1]
        mov     DWORD PTR [rdx], eax
```

However, once I began looking at the assembly I realized this was still doing some duplicate work. I then decided to try and utilize `memset` to solve the problem, though eventually switched to `wmemset` as will be discussed in the following section. This provided a speed improvement from around 7.9x over the unoptimized implementation, to 10.9x the unoptimized subroutine.

1.1.1 Challenges

`memset` only works on bytes, therefore `wmemset` has to be used to work with 4 bytes at once. This is a hacky and non-portable solution, as `wchar_t` is 4 bytes on Linux/Mac, and only 2 bytes on Windoze.

1.2 Array Initialization

Array initialization is a typical serial loop with no dependencies across iterations, therefore the most straightforward optimization is to unroll the loop to take advantage of the ILP available within our processor. I decided to unroll the loop 16 times, though didn't test values smaller or larger than 16. In addition to loop unrolling I am also prefetching 64 bytes ahead of where we're working at the top of each loop, allowing the cache to hopefully be filled by the time we move to the next row.

In addition to the above major optimizations I replaced all constants and math operations with efficient left shifts by immediate values, something the compiler would do for us if optimizations were enabled.

1.2.1 Challenges

Because the loop is manually unrolled we need to ensure that input lengths not divisible by 16 are handled correctly. This is often referred to as the "strip-mining" technique, with one main loop performing the unrolled body, and a second smaller loop handling the remainder in a scalar manner.

1.3 Factorial

Factorial is once again a serial loop (after rewriting the recursive call into a loop), through this time we have a carry-over dependency, meaning loop unrolling would still create a serial chain. However, I was able to achieve a significant speed improvement just by using the register keyword on all local variables. This allows for the branch comparison and accumulator multiplication to avoid writing/reading to/from memory. Again I prefetch the result memory address before the loop begins to ensure it is resident in cache.

1.3.1 Challenges

There were no real challenges for this implementation, through I avoided loop unrolling which could have added additional complexity.

1.4 Matrix Multiplication

I applied the same rough set of optimizations to matrix multiplication, namely the `register` keyword, and memoizing matrix index computations. These two optimizations alone were enough to grant around a 2.0x speed up.

From there I reordered the 2 inner loops to have better memory access patterns, in the hopes to exploit the spatial locality of the problem to achieve a further speed up. This final optimization gives a total speed up of 3.0x over the unoptimized implementation.

1.4.1 Challenges

Reordering the 2 inner loops required changing where we zero out the correct cell(s) of the result matrix. To achieve correct behavior I use a `memset` to 0 on each row the result matrix.

2 Comparison of Results

2.1 `-O0`

When compiling with `-O0` my programs beat the reference executable on all 4 benchmarks. The largest delta is for matrix initialization with the reference benchmark showing a 2.65x speed up and my optimizations yielding a 11.1x speed up. Our closest delta is in the array initialization, 3.38x for the reference, and 5.1x for mine. This makes sense as my manual loop unrolling can only improve the performance up until the processors' throughput limit.

2.1.1 `copt_O0`

```
jmowry4:hydra4 ~/copt> bash test.sh copt_O0
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):       16216.0
OPTIMIZED(ms):          1467.0
SPEEDUP:                  11.1

Running ARRAY_INIT with n = 300000 loop = 20000
```

```
UNOPTIMIZED(ms):       20983.0
OPTIMIZED(ms):          4100.0
SPEEDUP:                   5.1

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):       24600.0
OPTIMIZED(ms):          3766.0
SPEEDUP:                   6.5

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):       31700.0
OPTIMIZED(ms):         10617.0
SPEEDUP:                   3.0
```

2.1.2 copt_O0_ref

```
jmowry4:hydra4 ~/copt> bash test.sh copt_O0_ref
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):       16216.0
OPTIMIZED(ms):          6117.0
SPEEDUP:                  2.65

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):       20966.0
OPTIMIZED(ms):          6200.0
SPEEDUP:                  3.38

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):       23533.0
OPTIMIZED(ms):         10083.0
SPEEDUP:                  2.33

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):       32967.0
```

```
OPTIMIZED(ms):         21384.0
SPEEDUP:                  1.54
```

2.2 -O3

As I expected, turning optimizations up to -O3 negates the performance benefits from my optimized solutions. This is because my optimizations have likely made the compilers job harder, as it can no longer work with straight forward, easy to analyze loops.

It appears that the reference solution observes some of the same pitfalls as my optimizations, only deviating on array initialization and matrix multiplication. My array initialization code manually unrolled a loop 16 times, which likely hindered the compilers' ability to optimize this section of code. My matrix multiplication code did not use common optimizations like chunking, thereby making little to no difference in the generated assembly when optimizations were enabled.

2.2.1 copt_O3

```
jmowry4:hydra4 ~/copt> bash test.sh copt_O3
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):        1516.0
OPTIMIZED(ms):          1517.0
SPEEDUP:                   1.0

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):         933.0
OPTIMIZED(ms):          3983.0
SPEEDUP:                   0.2

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):        2050.0
OPTIMIZED(ms):          2016.0
SPEEDUP:                   1.0

Running MATRIX_MULTIPLY with n = 1600 loop = 1
```

```
UNOPTIMIZED(ms):        7150.0
OPTIMIZED(ms):          7183.0
SPEEDUP:                   1.0
```

2.2.2 copt_O3_ref

```
jmowry4:hydra4 ~/copt> bash test.sh copt_O3_ref
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):        1550.0
OPTIMIZED(ms):          1517.0
SPEEDUP:                  1.02

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):         933.0
OPTIMIZED(ms):          1700.0
SPEEDUP:                  0.55

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):        1966.0
OPTIMIZED(ms):          1917.0
SPEEDUP:                  1.03

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):        7216.0
OPTIMIZED(ms):          1617.0
SPEEDUP:                  4.46
```

2.3 What Optimizations are Effective and Where?

In general the most effective optimization (when compiling with `-O0`) is the use of the `register` keyword. When the compiler generates unoptimized code it typically stores local variables on the stack, which leads to unnecessary load/store instruction. If we instead instruct the compiler to leave these local variables in registers we can achieve a large speed-up. However, this optimization is far less effective under some real-world conditions. One example is a subroutine that uses a large number local variables, which would cause the compiler to “spill” our `register` values onto the stack. Another situation where `register` would not be helpful is when compiling with higher levels of optimization, as the compiler is able to apply this transformation automatically. The use of `register` helps when local variables are accessed frequently, such as arithmetic accumulators, or loop iterators.

The next key optimization is to remove naive uses of arithmetic operators, and fold in compile-time known constants. When compiling with optimizations the compiler can generate the optimal sequence of instructions for code of the form `x = y * 8`. However, without optimizations the compiler will sometimes choose to use an `imul` instruction directly. On 10th generation Intel CPU we can execute at most 1 `imul` instruction per cycle, whereas up to 2 `shl` instructions can execute per cycle (assuming our operands are of type register, and immediate, as they are in this example). Having said that, the optimal assembly sequence is neither an `imul` nor a `shl`, it would instead be a `lea` instruction. Unlike `imul` and `shl`, `lea` execute on multiple ports, allowing for more ILP, while maintaining the ability to execute 2 independent computations in the same cycle.

Lastly, the most complex optimization that I performed is loop unrolling, and transforming recursive functions into loops. This is a more complex optimization as it requires "bookkeeping" to ensure the results are consistent with the non-unrolled version. Loop unrolling is a common optimization that compilers will make when using higher levels of optimization, so performing it manually at `-O0` is a good choice. At a high level this optimization allows the CPU to execute more instructions at once by keeping the pipeline full, and reducing the total number of branch instructions.

2.4 Why Does `-O3` Hurt Some Optimizations?

My array initialization optimization was the only one hurt by enabling `-O3` optimizations, and I believe this comes down to the manual loop unrolling. My manually unrolled loop generates a complex assembly routine, totaling ~400 instructions. The naive implementation generates a sequence of only ~40 instructions, with the compiler only choosing to unroll the loop 2 times, while using a vectorization width of 4, resulting in 8 indices processed per loop iteration.

While large unroll widths can be helpful, they can also hurt performance as they generate more instructions, placing more pressure on the L1i cache, and potentially the CPU's own op cache.

2.5 What Optimizations Are Unaffected or Benefited by `-O3`?

One optimization that is unaffected by `-O3` is the use of `register`, which makes sense as the compiler is more than smart enough to automatically mark local variables as `register` without the explicit direction of the programmer.

Another optimization would be the transformation from a recursive function into an iterative loop, such as the factorial subroutine. Looking at the generated assembly for the unoptimized version compiled with `-O3` we see that the compiler correctly identified that it could transform this tail recursive function into an iterative one.

None of my own optimizations benefited from switching to `-O3`. The only optimization that I expected to benefit from `-O3` was the reordering of loops in the matrix multiplication, however this did not turn out to be the case. Looking at the assembly generated it turns out that the compiler was able to make the optimization for us, thereby negating any benefit from my manual reordering.