COSC561 – M9.10 PROGRAMMING ASSIGNMENT 5 - COPT

Giuseppe Schintu

University of Tennessee, Knoxville

COSC561 – M9.10 PROGRAMMING ASSIGNMENT 5 - COPT

On the first inspection into the routines to initialize matrixes and arrays, compute factorials and matrices operations, I noticed the check() and set() functions that seemed redundant for the operation performed. However, I decided to work on strength reduction and instruction selection optimizations first. This meant to move around code from inner loops to outer loops to reduce the number of calculations and their final assignment; in the matrix_initialize_opt, and array_initialize_opt routines, I noticed an increased speedup of 1 point just by using a couple of basic optimizations. At this point, I had to start researching for techniques and algorithms to address further optimizations. I ended up reviewing many different types of optimization techniques, some dependent on hardware and some only feasible for very large data. Perhaps the most influential that seemed to have a significant impact in my test runs were the loop unrolling and loop tiling.

Results with my run with -O0 flag

```
[gschintu:hydra0 /local_scratch/copt> ./test.sh copt_O0
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):       15150.0
OPTIMIZED(ms):          5016.0
SPEEDUP:                   3.0

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):       19666.0
OPTIMIZED(ms):          4684.0
SPEEDUP:                   4.2

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):       26133.0
OPTIMIZED(ms):          4167.0
SPEEDUP:                   6.3

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):       30384.0
OPTIMIZED(ms):         14967.0
SPEEDUP:                   2.0
```

Results with my run with -O3 flag. We can observe that the Matrix and Array initialization are not gaining much benefit after manual optimization. Both routines have already been optimized for some tradeoffs vectorization, hardware and software workload using SIMD. Ultimately, there could be some degradation since the code has been already optimized.

```
[gschintu:hydra0 /local_scratch/copt> ./test.sh copt
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):        1533.0
OPTIMIZED(ms):          1550.0
SPEEDUP:                   1.0

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):        1066.0
OPTIMIZED(ms):          1600.0
SPEEDUP:                   0.7

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):        4750.0
OPTIMIZED(ms):          3416.0
SPEEDUP:                   1.4

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):        6816.0
OPTIMIZED(ms):          3500.0
SPEEDUP:                   1.9
```

Results with REF program for comparison.

```
[gschintu:hydra0 /local_scratch/copt> ./test.sh copt_O0_ref
Running MATRIX_INIT with n = 3000 loop = 200

UNOPTIMIZED(ms):       15116.0
OPTIMIZED(ms):          5717.0
SPEEDUP:                  2.64

Running ARRAY_INIT with n = 300000 loop = 20000

UNOPTIMIZED(ms):       19600.0
OPTIMIZED(ms):          5816.0
SPEEDUP:                  3.37

Running FACTORIAL with n = 20 loop = 200000000

UNOPTIMIZED(ms):       21950.0
OPTIMIZED(ms):          9416.0
SPEEDUP:                  2.33

Running MATRIX_MULTIPLY with n = 1600 loop = 1

UNOPTIMIZED(ms):       30234.0
OPTIMIZED(ms):         20734.0
SPEEDUP:                  1.46
```

Array Initialization

After some basic optimization, it seemed logical to focus on either loop unrolling or loop tiling techniques. After testing both, it quickly became evident that loop unrolling was going to offer the most optimization as we could leverage the memory access allocation while decreasing the iterations. Past the 8 instructions, there was a decrease in performance, probably due to the code being larger than the instruction cache of the processor.

Matrix Initialization

In general, for the matrix initialization, most of the online information narrowed down to optimize a matrix initialization function by using pointer arithmetic instead of index-based access and by leveraging loop tiling to improve cache locality. However, most of the techniques and rewrites around cache locality, loop tiling and unrolling did not achieve much optimization at all. After some more research, I came across a SIMD lecture in regards of hw-level data parallelism, https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/SIMD.pdf .

It took a few days to rewrite an adaptation of the original code, and ultimately, I did not gain a dramatic optimization, but I was able to get it stable at over 10% of the REF run.

Matrix Multiplication

Like the matrix initialization, most of the online information suggested to optimize a matrix multiplication function by using pointer arithmetic instead of index-based access and by leveraging loop tiling to improve cache locality.

I added loop tiling with a TILE_SIZE of 8 initially, which I later increased to 16 as there was an increased .3 performance when running on Hydra server. This optimization can improve cache locality and overall performance. It is important to notice that this optimization will be more effective for larger matrices.

Additionally, I replaced the set function calls with direct pointer access to eliminate any possible overhead of the function call. I favored the optimization instead of readability.

Average tests run on Hydra

(8 TILE) ./copt 3 1600 1	./copt_O0_ref 3 1600 1	(16 TILE) ./copt 3 1600 1	./copt_O3_ref 3 1600 1	./copt_O3 3 1600 1
1.9	1.47	2.1	3.05	2.1

In conclusion, my optimization when using -O0 is approximately 30% faster than the REF. However, when using -O3, my optimization doesn't seem to gain additional advantage, but the REF_O3 does.

Factorial Recursive Routine

One way to optimize the factorial function is by using iteration instead of recursion. With a few more changes, we can use an accumulator to store the result and reduce the number of multiplications. And finally, we can also use a loop unrolling technique which ultimately decreases the overhead of the loop by decreasing the iterations.

Average tests run on Hydra

./copt 2 20 200000000	./copt_O0_ref 2 20 200000000	./copt_O3_ref 2 20 200000000	./copt_O3 2 20 200000000
5.4	2.30	1.1	5.4

This optimized version should have an increased 90% performance in both -O0 and -O3 because it eliminates the overhead of recursive function calls, makes better use of the cache and it computes the product of four consecutive numbers at a time, reducing the number of iterations by a factor of 4.

Conclusion

This assignment has been particularly rewarding. On one hand it was stressful to figure out new code and the SIMD instructions for the Matrix Initialization; all to get to a stable optimization. However, on the other hand, learning about optimization techniques, useful algorithms and having gained valuable experience in optimizing some basic code that is the baseline for most programs is very rewarding. After running our version of code optimization for multiple runs, we have obtained an average speedup optimization ranging from 11% to 150% in some operations.

	Matrix Init	Array Init	Factorial	Matrix Mult
My Opt	3	4	6	2
Ref Opt	2.7	3.4	2.32	1.55
	11%	18%	159%	29%