ASL HW2
Student Name: Berke Egeli
Legi No.: 20 948 220

Exercises:

2. Optimization Blockers (30 pts)

(a) Create a table with the runtime numbers of each new function that you created (include at least 2). Briefly discuss the table explaining the optimizations applied in each step.

(b) What is the speedup of function maxperformance compared to slowperformance1?

(c) What is the performance in flops/cycle of your function maxperformance.

Solution:

	Impl 1	Impl 2	Impl 3	Impl 4	Impl 6	Impl 7	Impl 13	Impl 18
Runtime (cycles)	53816.4	53460.1	49442.3	47504.9	8399.6	5295.25	3022.04	2517.28

Table 1. Runtime of different implementations in cycles

(a) Impl. 1 is the base implementation. Impl. 2 applies scalar replacement which by itself doesn't change the performance. Impl. 3 unrolls the loop by 6 iterations. The performance increases slightly because we exploit the ILP. Impl. 4 applies code motion and factors out the common operation in the sin evaluation outside of the loop which included the slow division operation. Eliminating 2 operations from each loop decreased the runtime somewhat. Impl. 6 exploits the fact that sin is a periodic function, so, we combine this idea with Impl. 3 remove the function from the loop. This stage also applies strength reduction to remove division operation from the inner loop by multiplying array elements with M_SQRT1_2 instead of dividing them with M_SQRT2 which yields the same result. At this point the only costly operation left inside the loop is the modulo operatör to get the item from z array. Impl. 7 does two things: First it converts every value inside the loop into a single precision floating point value, second it unrolls the loop by 8 which introduces an extra modulo operator to calculate the precomputed sin values which are now stored inside an array. Former is why the runtime decreases drastically while the latter is just a bad idea. Impl. 13 improves upon Impl. 7 by unrolling 12 times. This optimization realizes that sin values have a period of 3 iterations and z[] address calculation has a period of 4 iterations, so, by unrolling 12 times we can avoid using modulo operator to get the desired values for sin (from the sin array created in the Impl. 7 to precompute sin values) and z[] elements (we only Access z[0], z[10], z[20] and z[30]). We store all accessed z elements inside variables and we only calculate sin(60) because one of the sin values we calculate is 0.0 and the other is -sin(60). We remove all calculations that used the sin(0) since the result will be 0.0 anyways. We also break the load-compute-store pattern to store the final x value directly in the array rather than in a variable in compute phase then inside the array in the store phase. This probably prevents unnecessary accesses to variables. In this stage, all the operations inside the main loop are additions and multiplications. Impl. 18 finally removes all unnecessary array accesses from the inner loop. The assignments we make to the x array are of the following form: x[i] = (x[i] + sin(2 * M_PI * i / 3) * y[i] + z[(i%4) * 10]) * M_SQRT1_2 + y[i] * C1. This calculation is completed in 3 iterations without unrolling and in each iteration we make extra calculations to update the value of x[i] inside the loop. Notice we can rewrite this equation as the following: x[i] = x[i] * M_SQRT1_2 + y[i] * (sin(2 * M_PI * i / 3) +C1) * M_SQRT1_2 + z[(i%4) * 10] * M_SQRT1_2 . (sin(2 * M_PI * i / 3) +C1) * M_SQRT1_2 and z[(i%4) * 10] * M_SQRT1_2 can be calculated outside the main loop, eliminating the extra calculations inside the loop. This final optimization achieves the final cycle value.

(b) According to the measurements from the table the speed up of Impl 18 (maxperformance) compared to Impl 1 (slow_performance1) is 21.37, however, according to the results of code expert (which are obviously rigged against me) maxperformance implementation achieved a speed up of 20.21 against the base slow_performance1 implementation.

(c) Implementation 18 performs $\frac{52}{12}n + c$ flops where n is the size of the input array and c is a constant. The performance for n = 1024 is equal to ≈ 1.76 flops/cycle.

3. Microbenchmarks(40 pts)

(a) Do the latency and gap of floating point addition and division match what is in the Intel Optimization Manual? If no, explain why. (You can also check Agner's Table).

(b) Based on the dependency, latency and gap information of the floating point operations, is the measured latency and gap of function f(x) close to what you would expect? Justify your answer.

(c) Will the latency and gap of f(x) change if we compile the code with flags -O3 -fno-tree-vectorize (i.e., with FMAs disabled)? Justify your answer and state the expected latency and gap in case you think it will change.

Solution:

	Addition	Division	Division (min)	foo	foo (min)
Latency (cycles)	3.93	13.74	13.03	18.04	17.04
Gap (CPI)	0.49	3.99	3.99	4.01	4.01

Table 2. Microbenchmark Measurements of gap and latency for addision & division operations, and foo function

(a) The latency and gap values match the Intel Optimization Manual.

(b) The benchmark measurements are close to what I expected. The foo function uses a division and an FMA operation. The output of FMA operation is an input of the division operation. FMA has a latency of 4 cycles and a normal division operation has a latency of 14 cycles and a minimum latency of 13 cycles. So, the results I got 18 for normal foo and 17 for minimum foo match the expectation. For the gap, division operation has a gap of 4 cpi and FMA has a gap of 0.5 cpi. These operations can be issued in different ports, so, therefore the division operation is the bottleneck and the gap of foo is 4 cpi, same as division.

(c) I expect the latency of foo to increase now as we now have an extra operation in the critical path. FMA operation is split into 2, so, we have to multiply first then do the addition before we put that value as input into division operation, thus, I expect the new latency to be 22 cycles (21 cycles for minimum). I don't expect the gap to change. The division can only be issued in a single port. This leaves the addition and multiplication operations with one port as well. However, since they both have 0.5 cpi gap (Though, since division occupies one port it is more like 1 cpi) division gap is still larger than the gaps of addition and multiplication added, so, division is still the throughput bottleneck and the gap is still 4 cpi.