ASL Assignment 2

Fabian Wüthrich

March 13, 2020

1. Short project info (10 pts)

   Done

2. Optimization Blockers (25 pts)

   (a) The following table shows the run time of each iterative improvement.

base line	$2.49 \cdot 10^7$ cycles
inlining	$1.31 \cdot 10^7$ cycles
strength reduction	$8.08 \cdot 10^6$ cycles
remove aliasing	$7.93 \cdot 10^6$ cycles
1x loop unrolling	$4.09 \cdot 10^6$ cycles
max. performance	$1.97 \cdot 10^6$ cycles

Table 1: Runtime optimizations on a Intel Xeon Silver 4210 processor

   The simplest optimization was to inline the `compute()` function, which removes the overhead of a function call and already gives a 2x speed-up.

   As we are allowed to rewrite expressions using rules of exact arithmetic, we simplified several expressions and moved costly computation (e.g. inverse square root) out of the inner loop. There is still a slight improvement but it is not as good as inlining.

   In order to help the compiler we tried to remove aliasing by using a local variable for the computation in the inner loop, which then get hopefully assign to a register. The improvement was only minimal but we still used local variables as it is good practice.

   The next big considerable improvement was unrolling the inner and the outer loop once. The unrolling removes the branching in the inner loop, which was particularly slow as the correct branch cannot be predicted accurately.

   The last step was several unrollings of the outer loop. Now the computation can make optimal use of all ports and ILP comes in to our advantage. Unrolling the inner loop with accumulator did not give a significant improvement and increased the complexity of the code unnecessary. Similarly, unrolling the outer loop more than eight times did not increase performance so we finished optimizing at that point.

   (b) The improved function is **13x** faster than the base line.

   (c) The outer loop performs $\frac{n}{8}$ iterations. Then we use 32 operations to precompute the values from w. The inner loops performs $\frac{n}{2}$ iterations. In the inner loop body we execute 48 operations. The total number of operations is therefore

$$\frac{n}{8}(32 + \frac{n}{2} \cdot 48)$$

   The benchmark sets $n = 1000$ so we get

$$\frac{3'004'000\, operations}{1.97 \cdot 10^6\, cycles} \approx 1.5\, flops/cycles$$

3. Microbenchmarks (45 pts)

   (a) The Intel Optimization Manual lists a latency of 18 cycles and a gap of 6 cycles for the `SQRTSD` instruction on a Skylake architecture. This values coincide with our measurements. This is as expected because we time only one instruction without dependencies and all data fits into L1. Therefore, the CPU can distribute the operations optimally and we hit the theoretical limit.

   (b) The *sigmoid1* function executes first an multiplication (`MULSD`), then an addition (`ADDSD`), a square root operation (`SQRTSD`) and a final division (`DIVSD`). As every operation depends on the result of the previous one, the CPU has to wait until each instruction has finished. The following table shows the expected latency/gap given by the Intel Optimization Manual.

	Latency	Gap
`MULSD`	3	0.5
`ADDSD`	4	0.5
`SQRTSD`	18	6
`DIVSD`	14	4
Total	**39**	**11**

   We measured a latency of 30-36 cycles and a gap of 8-10, which is a bit faster but still close to the expected values.