CS/ECE 752
Fall 2024
Homework 6 SOLUTION
Due 11 AM Central Time on Friday, November 1st, 2024

Note: Several of you used X[i] = a*X[i] instead of the intended Y[i] = a*X[i]. This affects the answers below. Moreover, since this assignment is mostly about demonstrating the ability to run tests in HTCondor, the answers are shorter than usual below.

Problem 1 [7 points]

In your report, for the region of interest only, report the breakup of instructions for different op classes for each application and provide a brief analysis of the breakdown across all 6 applications. For this, as before, grep for the appropriate stats class for MinorCPU in the stats.txt file.

Solution:

All algorithms make significant use of the Integer ALU, presumably for the loop counter, and it accounts for between 37 and 54% of the total instruction count (for O0). This percentage goes down with O3 optimization, as the loop gets optimized and unrolled. Thus, the number of instructions accessing IntALU decreases.

Memory operations play a prominent role the instruction counts in all algorithms, accounting for between 30-40% of the operations, with the `*py' algorithms having the most. As expected, the daxpy, dax, saxpy, and sax have floating point instructions in their mix while the iaxpy and iax do not. In all algorithms, for the actual multiplication (and sometimes addition) calculations being performed, we can see the exact number of operations matching the problems size, e.g. when N = 64K, we see daxpy using exactly 64K floating point additions and exactly 64K floating point multiplications.

IAX/IAXPY use a = 2. As a result, the compiler optimizes this integer multiply into a shift of 2, which should both affect instruction count (SimdShift) and potentially improve performance. Moreover, In the iax* and sax* algorithms, we see the appearance of SIMD instructions, lowering the arithmetic operation counts by a factor of 2.

Somewhat surprisingly, the optimized code for iaxpy and iax now includes FloatMemRead and FloatMemWrite instructions. Inspection of the assembly reveals use of movdqu to load memory in 8-byte chunks, and then re-interpret the loaded memory as two 4-byte integers, thus saving on memory load operations.

Problem 2 [7 points]

In your report, again for the region of interest only, compare and contrast how many cycles it takes MinorCPU and O3CPU to simulate each of the 6 applications. Why do you think one CPU is faster than the other one? Provide statistical evidence based on your simulation results.

Solution:

In general O3 should outperform Minor. When looking at the achieved instructions-per-cycle (IPC) of each algorithm on each CPU, we see that the MinorCPU cannot `hide' its memory latencies with useful instructions the way that the out-of-order O3CPU can. In particular for the *AXPY programs, you likely saw that the O3 model had enough independent instructions (e.g., from unrolling and/or from independent loop iterations) that it could hide some of the memory access latency. Whereas MinorCPU being in-order makes it hard for it to do this.

When comparing O3 runs, you may have found that memory stalls contributed significantly. For all applications, regardless of single or double precision, you should see the L2 missing frequently because the working set is larger than the L2 size. However, the access pattern is relatively predictable, so next-block prefetching and spatial locality should allow for some hits. In particular for *AX, if you used Y = a*X, you should have the same memory footprint as the *AXPY workloads.

Problem 3 [7 points]

In your report, for the O3CPU runs you obtained in step 2, compare the performance of the region of interest across all 6 applications. Which applications perform the best? Why do you think they perform the best? Again, provide statistical evidence from your simulation results.

Solution:

Looking at O3CPU.py, the O3CPU uses the default functional unit pool, which has only a single IntALU, IntMultDiv, PF ALU, FP MultDiv, and SIMD Unit (plus memory read/write ports). The latency on integer multiplication is 3 cycles, while floating-point multiplication is 4 (see FuncUnitConfig.py), explaining why the integer program (un-optimized) sees a performance boost. Interestingly, all of the SIMD operations seem to have a latency of only 1 cycle -- somewhat unrealistic, but it definitely explains the jump in performance when SIMD instructions are used in the compiler optimized integer and single-precision floating point programs. When data is available in the L1 cache, the latency is very short (tag=2, data=2, response=2), so out-of-order execution hides this retrieval pretty well. Trips to L2 cache are minimum 20 cycle latency. Our L1 and L2 caches seem to both be configured by default with a BasePrefetcher() (see Prefetcher.py). The very regular access pattern of our data should be well accommodated by even the simplest prefetching algorithm to hide latency from L2 and main memory loads. However, since the working sets are large, this may mean we still have frequent misses.

Assuming you used Y = a*X, you should have seen that IAX was the fastest, because of the aforementioned reasons.