Morgan Rae Reschenberg
11 April 2018
CS 152
Laboratory Exercise 4 - Undirected Portion

4.2 Contest: Vectorizing and Optimizing Radix Sort

For the undirected portion of this lab, we elected to vectorise and optimise radix sort. Apart from two instances, we were able to vectorise rsort by using the RISC-V vector instruction extension to process *vector-length* number of elements per operation instead of a single element as would be done traditionally. First, we loaded a chunk of elements from our input vector, then we applied shift and remainder operations to obtain their least-significant digits. From there, we wanted to simply do a scattered store to our buckets vector to update the number of each LSB we'd seen, however we ran into an issue with concurrent writes. Because the vector extension processed all items from the same read of data, duplicate items weren't counted in our update.

To fix this, we created a vector of all 1's that we called our progress vector. Within a loop, we used <*vmfirst*> to get the index of the next element to process. Then, we used this index to access our LSB vector (the numbers we wished to count and use to increment our buckets). After extracting the element from our LSB vector, we created a scalar vector with this number. We then applied <*vseq*> to this scalar and our original LSB vector to create a mask which marked all the duplicates for this LSB. Again, using the mask, we ran <*vmpop*> to count the number of duplicates and, once again referencing the original LSB value extracted to the scalar, we inserted the count into a vector which kept track of our running sums. Finally, we subtracted the mask from our progress vector (as to not process elements twice), checked that the progress vector was not zero (e.g. there were items left to process) and jumped back to the beginning of the loop. After this, we were able to add our running sums vector to buckets and store our updated counts in memory.

Later, we ran into a similar issue with trying to store our items from the input array into the final scratch array using the index vector we'd created. We noticed that trying to store items to the same bucket resulted in all items being stored at the same index (i.e. all elements were overwritten by the last element in that bucket). To fix this, we employed a technique similar to the one above. We again created a progress vector, extracted elements one-by-one, masked and counted the duplicates, and decremented the progress vector by the mask to avoid processing more than necessary. Within the body of the loop, though, we created a vector to subtract instead of a vector to add. As we iterated through the mask, we subtracted one from the original index for each duplicate we'd counted thereby generating a new index array which gave distinct indices to each element in the same bucket. After getting this information, we were able to do a scattered store at the new indices from the current chuck of our input array.

In terms of optimisations, we expected to see a slight speedup due to the vectorisation (e.g. processing <*VL*> elements per operation instead of one), but knew that we could optimise further by unrolling our code as we'd seen done in <rsort.c>.

Before unrolling, we acquired the following statistics and speedup from our vectorised code.

	rsort
CPI	2.061849103
Vector (CPI)	1.109686193
Speedup (CPI)	1.858047002

First, we unrolled the to_buckets loop which increased the amount of data we grabbed from our input vector, found the LSB's of, and added to our partial sums. This decreased our overall CPI by decreasing the amount of cycles required.

	vec-rsort	**vec-rsort (to_buckets unrolled)**
cycles	266457	253596
instructions	234316	237823
D$ accesses	118639	118639
D$ misses	128	128
CPI	1.1371694635	1.0663224331

Next, we worked on unrolling the from_buckets loop in addition to our to_buckets loop. In from_buckets, we pulled in twice as many partial sums, found our duplicate items, and calculated the indices to store. We found that unrolling this loop in addition to the to_buckets loop decreased our overall CPI but increased our cycles. We attribute the increase in cycles to the nature of the code written; in from_buckets we do a lot of conditional processing. For each duplicate found, we have to iterate through all matching duplicates and update the index vector duplicate_count times. When increasing the amount of data we process, that duplicate_count has potential to increase depending on the data we're processing. Because the amount of times we have to execute that loop is data dependent rather than dependent on the amount of data, unrolling doesn't help us increase the speedup.

	vec-rsort	**vec-rsort (to_buckets unrolled)**	**vec-rsort (to_buckets, from_buckets unrolled)**

cycles	266457	253596	282425
instructions	234316	237823	271954
D$ accesses	118639	118639	118639
D$ misses	128	128	128
CPI	1.1371694635	1.0663224331	1.038502835

Our final CPI for this project ended up being roughly 1.04. We unrolled each loop two times and, if we had more time, would look into removing costly vector instructions, unrolling further to eliminate short loops, and loading/storing to memory less frequently (e.g. keeping a buckets vector in one of our vector registers throughout the program).