1 Introduction

Since MMX was first introduced in the 90's, x86 CPUs have offered an growing array (I'm not sorry!) of vector functionality. In this homework, we will experiment with one of the more recent vector extensions, AVX2, which offers many powerful packed integer instructions, operating on 256-bit registers. More specifically, the assignment consists of variations of the problem of finding the smallest value in an array.

2 Preliminaries

This homework requires access to a machine with AVX2, which is only available on our largest research server:

nodes `ssh nodes.cs.uic.edu` 128 Intel Broadwell threads

your UIC netid password applies. For this homework, we are writing single-threaded programs, however, so if your own computer supports AVX2, feel free to use that. Even (shudder) virtual machines could be ok, as long as the host CPU supports AVX2. Watch out: if a researcher is running parallel code on nodes while you are running your program, your performance results could be heavily affected. Before running a performance experiment, take a quick lock with `top`.

3 Assignment

The template for this assignment is available at `https://github.com/uicperformance/minvec`. To build the programs, simply run `make` in the checked-out repository folder. For each assignment, only edit the file that contains the min-function in question, leave the main program alone.

3.1 Establish performance baseline, and prepare LaTeXreport

In `cmin.c`, complete the implementation of *arraymin()*, and *minindex()* so that `test_cvec` and `test_novec` complete without error. While *arraymin()* returns the minimum value, *minindex()* should return the index of the smallest value. Run `benchmark_cvec` and `benchmark_cnovec` to measure the performance of your implementation with and without vectorization.

Using gnuplot, generate a plot of benchmark_cvec and bencmark_cnovec (four separate, labeled lines), with input size on the x axis, and cycles/op on the y-axis. Add this to your LaTeX report, which you will use to generate your report.pdf. Use sizes from 8 to 1024 elements.

3.2 Iterative Vector Min

In `iterative.c`, implement a vectorized *arraymin()* function using a combination of C and inline assembly. You may assume that the input size is a multiple of

8 integers. Similar to what we did in class, use a C for-loop, and an inline assembly loop body using the VPMINSD instruction to produce a vector of 8 values, one of which is the smallest. Then, finish the job after the loop using another chunk of inline assembly, to compute the single minimum value. For this part, consider using a combination of the instructions VPSHUFD, VPERM2I128, VALIGNR.

Plot cycles/op vs. input size. How does this compare the vectorized C version? Study the assembly of the vectorized C version and try to determine why. Add this to your report.

3.3 Iterative Vector Min with Index

In `iterative.c`, implement *minindex()*, another vectorized min function using a combination of C and inline assembly. Here, VPMINSD will not help. Instead, use a combination of VMOVDQA (move), VCMPGTD (compare), VPAND, VPXOR, VPOR, VPADDD, and VPBROADCASTD, and perhaps others.

Start by writing a replacement for the VPMINSD instruction using VCMPGTD (compare), VPAND, and VPOR. Then, add instructions for tracking the index of each of the 8 array minima. Hint: VCMPEQD %%ymm0,%%ymm0,%%ymm0 sets every bit in *ymm0* to 1. VPBROADCASTD puts the same scalar in each integer-sized slot of the a vector register.

For simplicity, you may use C code to compute the final index outside the loop.

Plot cycles/op vs. input size. How does this compare the vectorized C version of minindex? Explain.

3.4 Faster, Fixed-Sized Vector Min (bonus)

In `fixed.c`, implement *arraymin64()*. Here, the array always has 64 integer elements in it. Write the fastest array min you can come up with, targeting Broadwell, using one big block of inline assembly, no branches, no loops. Here, feel free to make use of VPMINSD, and anything else you want to try.

Once `test_fixed` runs without error, announce your best `benchmark_fixed` performance on Piazza.

Hint: keep data dependencies, instruction latency and reciprocal throughput in mind. Check Agner Fog's tables for Broadwell. Report performance in cycles/op, and explain briefly how your solution works.

3.5 turn-in instructions

Similar to the previous homework, turn-in is by git classroom, using this invitation link `https://classroom.github.com/a/46nOdbWP`. Include all your code, and add `report.pdf` to the root folder of your submission.

3.6 Double-check your submission

To make sure that you submitted everything you think you submitted, *git clone* your turn-in repository into a fresh folder, check that your `report.pdf` is in the submission folder, and that all the Makefile targets build correctly.