Activity 3.0: Vectorization

This mini activity is designed to help you get more comfortable with vectorization. There are 3 total questions in two parts.

Due date: Thursday October 5th, 2023, 9:00 pm EDT.

Instructions for Submission: Submit via Gradescope.

Part 1: Reading Vectorized Code

In this problem you need to read, understand and explain what is going on in a few snippets of code that the compiler generated for the problem we discussed in class. You will want to look up each instruction to understand what it is doing and what the cost is.

The code and assembly can be seen https://godbolt.org/z/d3oKW4K3E

Question 1

We first look at the loop body

```
.LBB0_6:
        vpcmpeqd        xmm3, xmm1, xmmword ptr [rdi + 4*rax]
        vpmovzxdq       ymm3, xmm3              # ymm3 =
xmm3[0],zero,xmm3[1],zero,xmm3[2],zero,xmm3[3],zero
        vpand   ymm3, ymm3, ymm2
        vpaddq  ymm0, ymm0, ymm3
        add     rax, 4
        cmp     rcx, rax
        jne     .LBB0_6
```

Please explain what this is doing, and how many cycles each iteration of the loop takes.

Your explanation should include what are the inputs (what values are in each register at the beginning), what are the outputs (the values of the registers at the end), and how it is computing this. Your answer should be complete in that all 7 instructions must be explained.

`Response to Question 1:`

`.LBB0_6:` is the label for the loop body

[Instruction 1] : `vpcmpeqd        xmm3, xmm1, xmmword ptr [rdi + 4*rax]`

`vpcmpeqd` : Compare packed 32-bit integers in a and b for equality, and store the results in dst.

Here, it is used to make bit comparisons between the `inputs` `xmm1` (which is the vectorized form of `target` in the C++ code in lecture and `xmmword ptr [rdi + 4*rax]` (which consists for four 32-bit chunks of `data` starting at `rdi`(ptr to a memory location) and uses `rax` as the loop counter `i` for indexing for memory access in C++).

After comparing equality, the results is stored as `output` in `xmm3` (which all of the 32 bits can be all ones for true, or all zeros for false).

`vpcmpeqd` takes up one cycle.

[Instruction 2] : `vpmovzxdq       ymm3, xmm3                  # ymm3 = xmm3[0],zero,xmm3[1],zero,xmm3[2],zero,xmm3[3],zero`

`vpmovzxdq` : Zero extend packed unsigned 32-bit integers in a to packed 64-bit integers, and store the results in dst.

Here, the `input` `xmm3` gets zero extended from 32-bits to 64-bit and the results in stored in `ymm3` as `output`. Now `ymm3` holds the values in a 256-bit array.

In this case, if `xmm3` contains all 1's (vectorized target was equal to all of the elemends of `data`) then each of `ymm3` bits would consist of (upper half)[32 zeros] 32 ones.

`vpmovzxdq` takes up 2-4 cycles (based on modern intel processors in sept 2021 data)

[Instruction 3] : `vpand   ymm3, ymm3, ymm2`

`vpand` : Compute the bitwise AND of 256 bits (representing integer data) in a and b, and store the result in dst.

Here, `vpand` does bitwise AND operation between `inputs` the 256-bit register we just extended `ymm3` and another 256-bit register `ymm2` and stores results in `output` `ymm3`

`vpand` takes up one cycle

[Instruction 4] : `vpaddq  ymm0, ymm0, ymm3`

`vpaddq` : Add packed 64-bit integers in a and b, and store the results in dst.

Here, `vpaddq` adds the inputs `ymm0` and `ymm3` (both 256-bit registers) and stores the results in `output` `ymm0`. This means that after the loop terminates, we get the computed total count of how many occurences of a certain element there was.

`vpaddq` takes up one cycle

[Instruction 5]: `add rax, 4`

`add` increments the register of `rax` (the loop counter) by 4 in which is helpful to move to the next set of elements in the `data` array.

`add` takes 1 cycle.

[Instruction 6]: `cmp rcx, rax`

`cmp` compares the values of `inputs` `rax` to `rcx` (this in our `n` in C++) and sets a result to the processor status register and tells us if it is the end of the loop or not.

`cmp` takes 1 cycle.

[instruction 7:] `jne .LBB0_6` sees if the result of the two registers (from the previous instruction) are equal or not. If not equal, then it jumps back to the beginning of the loop body to do another set of the instructions for the next iteration. If equal, then the loop stops. Comparing and then jumping (or exiting) takes 1 cycle.

`INPUTS` : `xmm1` (target) `rdi` (ptr to memory access) `rax` (loop counter i) `rcx` (max size n)

`OUTPUTS` : `ymm0` (count) `rax` (increases by 4 after each iteration) `flag (processor status) register` (the results of comparison is updated and determines the termination to continuation of the for loop)

`SUMMARY` : Using vectorization, this loop counts the number of occurrences of an element/number (four numbers at a time) and stores it in an array with values for occurrences. The total cycles are around 8-10 given these seven intructions in the loop.

Question 2

After the loop

```
        vextracti128    xmm1, ymm0, 1
        vpaddq  xmm0, xmm0, xmm1
        vpshufd xmm1, xmm0, 238                 # xmm1 =
xmm0[2,3,2,3]
        vpaddq  xmm0, xmm0, xmm1
        vmovq   rax, xmm0
```

Please explain what this is doing, and how many cycles it takes.

Once again please include the inputs, outputs, and cost. You must include an explanation for every instruction and what it is doing.

`RESPONSE TO QUESTION 2:`

[Instruction 1] : `vextracti128    xmm1, ymm0, 1`

`vextracti128` : Extract 128 bits (composed of integer data) from a, selected with imm8, and store the result in dst.

Here, `ymm0` (the 256-bit array that represented the vectorized from of `count`) gets extracted 128-bits and stored in `xmm1`. It is selected with the argument imm8 of `1` which specifies which part of the 256-bit array it is extracting.

`vextracti128` takes up 3 cycles

[Instruction 2]: `vpaddq  xmm0, xmm0, xmm1`

`vpaddq` : adds the elements `xmm0` and `xmm1` registers, stores the results in `xmm0`(128-bit register)

`vpaddq` takes 1 cycle

[Instruction 3]: `vpshufd xmm1, xmm0, 238`

`vpshufd` shuffles the four 32 bit elements in `xmm0` using the shuffle pattern `238` and then stores the results in `xmm1`.

`vpshufd` takes up 1 cycle.

[Instruction 4]: `vpaddq  xmm0, xmm0, xmm1`

`vpaddq` : Add packed 64-bit integers in a and b, and store the results in dst.

Here, it will add the components of `xmm1` to `xmm0`, stored into `xmm0` (the rightmost 64-bits)

`vpaddq` takes up 1 cycle.

[Instruction 5]: `vmovq   rax, xmm0`

`vmovq` moves the rightmost 64 bit of `xmm0` into `rax` to return. This takes one cycle. This takes 2-3 cycles (based on Piazza)

`SUMMARY`: each iteration takes up 8-9 cycles.

Part 2: Writing Vectorized Code

In this part you will tackle a new problem, write some code for it, and then analyze it. The problem can be found at http://preview.speedcode.org/ide/index.html?count_pairs

The goal of the problem us to count unaligned pairs of bytes in an array.

The starting code is

```
uint64_t count_pairs(uint8_t *data, uint64_t size, uint8_t target) {
  uint64_t total = 0;
  for (uint64_t i = 0; i < size - 1; i++) {
    if (data[i] == target && data[i + 1] == target) {
      total += 1;
```

```
    }
  }
  return total;
}
```

Question 3

Please achieve 1,000% speedup or more over the reference code and include your code in your submission.

You must explain your solution in English as well. Submissions without a full explanation will not receive points.

If you did it using intrinsics then explain your inner loop as you did for the previous problem. Including:

- how does it compute the answer?
- how many cycles does it take?
- how many iterations of the base loop from the starting code does it compute on each iteration?

If you did it without using intrinsics please explain what you did to transform the problem into a form that the compiler could vectorize.

Yes, this is a hint that it can be done either with, or without intrinsics

`MY APPROACH WRITE-UP / EXPLANATION:`

In order to get greater speedup over the reference code, I chose to use `intrinsics` (11_vectorization.ipynb) and `cilk reducers` (06_fork_join.iypnb).

`_mm256_set1_epi8(target)` : `target` is `uint8_t` (8 bits) so the 256-bit vectorized register is able to process 256 / 8 bits = 32 elements iterations of the base loop from the starting code.

My inner loop includes the following:

`_mm256_loadu_si256((__m256i *)(data + i))` and `_mm256_loadu_si256((__m256i *)(data + i + 1))` : this gets our comparisons into two vector registers, loading 256-bits of integer data from memory into dst. mem_addr . Each takes up 7 cycles and it is called twice so, 14 cycles.

`__m256i curr32Elements = _mm256_cmpeq_epi8(curr32Elements, vecTarget)` : Compare packed 8-bit integers in `curr32Elements` and `vecTarget` for equality, and store the results in dst. *This is also the same logic for the other comparision with data[i + 1] and also comparison BOTH afterwards* Each takes one cycle.

`int movemask = _mm256_movemask_epi8(compareToPair)` : Create mask from the most significant bit of each 8-bit element in a, and store the result in `movemask`. Takes 2-4 cycles.

`total += _mm_popcnt_u32(movemask)` : Count the number of bits set to 1 in unsigned 32-bit integer `movemask`, and return that count in `total`. Takes 3 cycles.

Total cycles = 14 + 1 + 1 + 1 + 2-4 + 3 = `22-24 cycles`.

There will be at most 31 remaining elements to take care of using the reference for loop for computation. At the end of the second for loop, `total` would have counted all of the pairs that share the same target value.

`Results:`

make -f /sandbox/Makefile run SPEEDCODE_SERVER=1 correctness make -f /home/tmp/speedcode-proto/sandbox_utils/Makefile.common correctness make[1]: Entering directory '/box' /home/ubuntu/OpenCilk-2.0.0-x86_64-Linux-Ubuntu-20.04/bin/clang++ /sandbox/driver.cpp -c -fopencilk -O3 -march=native -gdwarf-4 `pkg-config --cflags catch2-with-main` -I/home/ubuntu/miniconda3/envs/speedcode-proto/include `/bin/python3 /home/tmp/speedcode-proto/sandbox_utils/get_copts.py3` -O3 -I/home/tmp/speedcode-proto/sandbox_utils/include -o /box/driver.o /home/ubuntu/OpenCilk-2.0.0-x86_64-Linux-Ubuntu-20.04/bin/clang++ /sandbox/solution.cpp -c -fopencilk -O3 -march=native -gdwarf-4 `pkg-config --cflags catch2-with-main` -I/home/ubuntu/miniconda3/envs/speedcode-proto/include `/bin/python3 /home/tmp/speedcode-proto/sandbox_utils/get_copts.py3` -O3 -I/home/tmp/speedcode-proto/sandbox_utils/include -o /box/solution.o /home/ubuntu/OpenCilk-2.0.0-x86_64-Linux-Ubuntu-20.04/bin/clang++ /box/driver.o /box/solution.o /home/tmp/speedcode-proto/sandbox_utils/lib/nanobench.o -fopencilk `pkg-config --libs catch2-with-main` -o /box/tmp LD_LIBRARY_PATH=:/home/ubuntu/miniconda3/envs/speedcode-proto/lib:/home/ubuntu/OpenCilk-2.0.0-x86_64-Linux-Ubuntu-20.04/lib /box/tmp [correctness] --reporter XML::out=/box/result.xml --reporter TAP::out=/box/result.tap --reporter console Filters: [correctness]

Randomness seeded to: 2627238558

All tests passed (2 assertions in 1 test case)

make[1]: Leaving directory '/box' make -f /sandbox/Makefile run SPEEDCODE_SERVER=1 benchmark make -f /home/tmp/speedcode-proto/sandbox_utils/Makefile.common benchmark make[1]: Entering directory '/box' LD_LIBRARY_PATH=:/home/ubuntu/miniconda3/envs/speedcode-proto/lib:/home/ubuntu/OpenCilk-2.0.0-x86_64-Linux-Ubuntu-20.04/lib /box/tmp [benchmark] --reporter XML::out=/box/result.xml --reporter TAP::out=/box/result.tap --reporter console Filters: [benchmark] Randomness seeded to: 2910530557

relative	ns/op	op/s	err%	total	benchmark
100.0%	454,693,355.00	2.20	0.0%	2.27	`Reference`
3,308.3%	13,744,034.33	72.76	0.4%	0.99	`Submission`
477.1%	95,297,024.50	10.49	0.9%	0.96	`vector serial`
695.7%	65,361,337.00	15.30	0.0%	1.11	`simple parallel`
3,101.6%	14,659,837.57	68.21	0.3%	1.00	`vector parallel`

===
test cases: 1 | 1 passed assertions: - none -

make[1]: Leaving directory '/box'

`MY VECTORIZED CODE FOR QUESTION 3`

In [ ]:

```
#include "solution.hpp"
#include <cilk/cilk.h>
#include <cilk/opadd_reducer.h>

uint64_t count_pairs(uint8_t *data, uint64_t size, uint8_t target) {
    cilk::opadd_reducer<uint64_t> total = 0; // reducers using cilk 06_fork_jo

    // find the end of the main loop which is divisible by 32
    uint64_t clean_end = (size / 32) * 32;

    // the target int, we already have it in one int,
    // we just need it in all locations in a vector register
    __m256i vecTarget = _mm256_set1_epi8(target);

    cilk_for (uint64_t i = 0; i < clean_end; i += 32) {
        // get our comparisons into two vector registers
        // loading the data from the array
        __m256i curr32Elements = _mm256_loadu_si256((__m256i *)(data + i)); //
        __m256i compareCurrToTarget = _mm256_cmpeq_epi8(curr32Elements, vecTarg

        __m256i leftShiftElements = _mm256_loadu_si256((__m256i *)(data + i + 1
        __m256i compareLeftToTarget = _mm256_cmpeq_epi8(leftShiftElements, vecT

        __m256i compareToPair = _mm256_and_si256(compareCurrToTarget, compareLe
        // gets the result back out of a vector register and into a normal int
        int movemask = _mm256_movemask_epi8(compareToPair);
        total += _mm_popcnt_u32(movemask); // count the bits in the mask
    }

    // iterate through left over bits
    for (uint64_t i = clean_end; i < size - 1; i++) {
        if (data[i] == target && data[i + 1] == target) {
            total += 1;
        }
    }
```

```
    return total;

}
```