Marco Montagna
22481726

CS194
HW1

Sum 1.3

With Printing not hardcoded
10 Trials with -O2 enabled resulted in an mean of 23,862 ticks.
10 Trials without -O2 enabled resulted in an mean of 106,897 ticks.

Without printing not hardcoded
10 Trials with -O2 enabled resulted in an mean of 3,840 ticks.
10 Trials without -O2 enabled resulted in an mean of 103,872 ticks.

Without printing Hardcoded Input
10 Trials with -O2 enabled resulted in an mean of 3,910 ticks.
10 Trials without -O2 enabled resulted in an mean of 104,154 ticks.

With printing Hardcoded Input
10 Trials with -O2 enabled resulted in an mean of 24,081 ticks.
10 Trials without -O2 enabled resulted in an mean of 103,850 ticks.

When printing is not enabled and O2 is, the compiler can eliminate the sum computation, as it's not used. Similarly when O2 is enabled and the input is hardcoded the compiler can eliminate the calculation and replace it with a constant; though in this particular case it doesn't seem to matter. In fact there seems to be a slight decrease in performance. I swear I checked it twice.
gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

atoi(argv[1]) converts the first command line argument to an integer. argv[1] is the first argument. atoi() does the conversion from the input string to an int.

Pointer Chasing 2.3

Each iteration of the loop depends on the result of the last, so it must wait until a[i] is loaded, for before it can proceed. We can divide the aggregate time needed to finish X amount of iterations by X to obtain the average number of ticks per load.

Some simple tests with the sizeof operator indicate that an int array of length N takes 4*N bytes, ie an int is 4 bytes.

See Attached

There do no appear to be any sudden slope changes in the 32KB to 8MB range. I was quite careful in my timing mechanisms, and each database is the result of averaging 1,000 runs. I did notice a very drastic change in slope at about 16KB. I've attached a graph to illustrate. This is probably about where the cache on my machine begins to be unable to provide 100% of results, going on past this point there is a nearly linear increase in cycles/step.

I don't know that we can be sure no pipelining is occurring, but we do know that since each step depends on the last, the instruction will stall somewhere in the pipeline process, before it loads the memory, until it's operands become available. But I don't see any reason why IF, ID, and similar stages couldn't be completed.

Bonus: Aren't we basically treating the array as a directed graph? So any algorithm that follows the graph starting at node 0, counting the number of steps it takes until it returns to the 0 node.

```
BONUS(array):
    i = 0;
    set = {}
    while (true) :
        i = a[i];
        if (not set.contains(i)):
            set.push(i)
        else
            break;
    return length(set)
```

Bandwidth 3.4
The chart was produced from data generated with a step size of 7000 bytes, running 100 tests per step.

The basic SIMD copy plot is nearly constant throughout the range of the tests. In all of the copy methods tested there appears to be a high frequency "noise", which is probably due to cache effects at particular array sizes, eg significant cache conflicts.

The cache SIMD method is initially capable of high bandwidth/throughput, but it quickly tapers off, presumably due to increasing cache misses. While there are two very observable dips in bandwidth, neither correspond with any of the cache sizes on my machine, (16KB, 2MB, 8MB).

Interestingly the naive method (my method), exhibits bandwidth comparable to that of the basic SIMD method. The naive method exibits a dip in performance right on the 5.2MB array size, which matches a similar dip in the cache SIMD method. I suppose this suggests some cache effect, but as already mentioned 5MB is not near to a cache boundary.

Warming up the cache, does just that it preloads some of the array to be copied into the cache, which saves us from measuring compulsory misses.

An inefficient array copy procedure would waste time on unnecessary computation, and as a result we would end up waiting for the memory to return results, which could have be loaded earlier. In effect we end up waiting on the latency rather than measuring the bandwidth.

- _mm_prefetch : fetches a cache line and moves it closer to the CPU in the memory hierarchy.
- _mm_load_si128 : loads a 128 bit value into a register, looks to be otherwise identical to a normal load.
- _mm_stream_si128 : stores directly without inserting into the cache, if the value is already present in the cache it is updates. Useful to prevent evicting prefetched vals etc..
- _mm_store_si128 : similar to load_si_128, simply stores a 128 bit value.

4 Flops

	naive_sgemm	opt_simd_sgemm	opt_scalar1_sgemm	opt_scalar0_sgemm
FLOPS	110,043,092	9,731,842,743	1,238,313,777	1,616,109,176
IPC	0.109252	1.474648	1.710477	0.792767

You didn't give us the code for the procedures, so we can't be sure what algorithm you are using but it should be O(n^3), most probably it's n^2 (2n − 1) floating point operations for the 2^10 matrices. Flops are calculated using the n^2 (2n − 1) number .

It's possible to have IPC greater than one on a superscalar machine, because the processor will be executing multiple instructions (or parts of them) at the same time.

Greater IPC, only indicates that more instructions were on average completed per clock. When comparing two different programs IPC is almost meaningless, because it does not account for what amounts to wasted instructions.