Jeremy Prater

CS 475

Project 5 – SIMD and Reduction

**Target Machine:**

I ran the test on flip1.engr.oregonstate.edu. The python script generates an executable using GCC 5.3 and CMake to generate the makefile. The python script then calls the executable with various array sizes using SIMD and reduction combinations.

> ./testBuild.py

To correct for other processes and workloads on the machine, each combination is executed 10 times and an average and best performance is recorded. Test results are available in

> cat ./build/project5.csv

**Table and graph:**

A selection of array sizes were picked to give a good spread of performance vs array size. 2 kilobytes to 256 megabytes were picked.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| useSIMD | doReduction | arraySize | megaOpsSecAvg | megaOpsSecBest |
| 0 | 0 | 268435456 | 1299.883603 | 1329.190622 |
| 0 | 0 | 33554432 | 1312.525389 | 1342.951944 |
| 0 | 0 | 2097152 | 1310.436777 | 1363.479989 |
| 0 | 0 | 262144 | 1320.045925 | 1374.799021 |
| 0 | 0 | 65536 | 1334.94795 | 1379.934878 |
| 0 | 0 | 2048 | 1241.666976 | 1363.523953 |
| 0 | 1 | 268435456 | 1190.173039 | 1193.052214 |
| 0 | 1 | 33554432 | 1191.115546 | 1192.693856 |
| 0 | 1 | 2097152 | 1204.467728 | 1212.880739 |
| 0 | 1 | 262144 | 1206.688809 | 1220.489027 |
| 0 | 1 | 65536 | 1214.083659 | 1222.390525 |
| 0 | 1 | 2048 | 1123.613129 | 1200.504029 |
| 1 | 0 | 268435456 | 3199.501756 | 3307.486055 |
| 1 | 0 | 33554432 | 3167.935029 | 3280.432762 |
| 1 | 0 | 2097152 | 4918.803451 | 5575.618154 |
| 1 | 0 | 262144 | 5040.449879 | 5725.545561 |
| 1 | 0 | 65536 | 5422.511428 | 7483.051355 |
| 1 | 0 | 2048 | 4835.675108 | 6871.947674 |
| 1 | 1 | 268435456 | 5103.29349 | 5128.060319 |
| 1 | 1 | 33554432 | 5036.135992 | 5135.427662 |
| 1 | 1 | 2097152 | 7380.07319 | 7444.969244 |
| 1 | 1 | 262144 | 7463.709998 | 7583.656017 |
| 1 | 1 | 65536 | 7581.286609 | 7720.103585 |
| 1 | 1 | 2048 | 5258.61961 | 6829.264769 |

**Speedup Patterns:**

Without SIMD a slight increase in performance as array size increases which peaks out around 1000 mops (millions of operations per second). When enabling reduction without SIMD, the performance almost identical, reduction incurs a moderate performance hit of roughly 10%. By unrolling the loop in the non-SIMD multiply function, cache alignment was preserved and a small increase in performance as noted. The previous data was lost, but the function was using a linear for loop and the performance was slightly lower than the reduction test.

With SIMD we see a much different picture. Both reduction and multiplication show an exponential increase until an array size of 8192 bytes. As we increase the array size, we see the prefetcher helping feeding the processor until the cache rate reaches the maximum. We see the simple multiplication performance begin to decrease until 8MB (8,388,608 bytes). cat /proc/cpuinfo shows that the processor has 12 MB of L3 cache. The slowdown could be caused by flushing cache memory back to main memory via L3 cache. We do not see the slowdown with reduction until array size is 4MB, but total performance is still higher.

**Are they consistent across a variety of array sizes?**

Yes, speed patters show consistency across array sizes and when comparing SIMD vs non-SIMD and reduction vs non-reduction.

**Why or why not, do you think?**

The program is demonstrating spatial and temporal coherency causing cache hits and increased performance. This is helped by the fact that a call to aligned\_alloc() was used to ensure program memory began on a 64-byte boundary to increase cache alignment.

**Knowing that SSE SIMD is 4-floats-at-a-time, why could you get a speed-up of < 4.0 or > 4.0 in the array multiplication?**

Getting a speed-up of less than 4.0 could be caused by cache exhaustion, poor prefetch performance due to small array size or other processes evicting cache lines reducing performance.

A speed up of greater than 4.0 is caused by efficient pre-fetching, which keeps the SIMD units ready to process data, increasing performance. Also, timing errors could cause artificially high measurements due to the small-time scale involved. I believe timing errors are minimized because the chart above was generated using an average of 10 samples per array size.

I chose two array sizes to compute speed up for non-reduction multiplication, 64KB and 32MB.

64KB Speed Up (Non Reduction)

32MB Speed Up (Non Reduction)

**Knowing that SSE SIMD is 4-floats-at-a-time, why could you get a speed-up of < 4.0 or > 4.0 in the array multiplication-reduction?**

As the same reasons, above, a speed up of less than 4.0 could be caused by poor prefetching due to small array sizes, and other processes interfering with cache lines. However, cache exhaustion is not as much of a concern since we are not writing data back into main memory, which saves half of the memory bandwidth.

Achieving a speed up greater than 4.0 is obvious here. Once array size reaches 1024 bytes (1KB), we reach greater than 4.0 speed up and the value does not dip below 4.0 after that. This can be attributed to not writing back to main memory and efficient cache prefetching. It is noticeable that reduction performance is generally higher than non-reduction performance, except in small data sets.

Like the non-reduction example above, I picked 64KB and 32MB as sample buffer sizes to compute a speed up factor.

64KB Speed Up (Reduction)

32MB Speed Up (Reduction)