1 Machine

Laptop running an i7-4750HQ @ 2GHz, turbo to 3.2 GHz. Turbo can be sustained for the duration of the program so fix frequency at 3.2. 4 cores, 8 threads. It can execute two FMA operations per cycle with up to 8 floating point elements. Therefore the per core performance is $8 \times 2 \times 2 \times 3.2 = 102.4$GFLOPS/s.

Cache wise we have the following:
L1 Data Cache: 32KB, 8-way associative, 64 byte line size
L1 Instruction Cache: 32KB, 8-way associative, 64 byte line size
L2 Unified Cache: 256KB, 8-way associative, 64 byte line size
L3 Unified Cache: 6144KB, 12-way associative, 64 byte line size
L4 Unified Cache: 131072KB, 16-way associative, 64 byte line size (This is vram).

Memory wise we have dual channel DDR3-1600 memory, each tick we can fetch 64 bits which comes down to a bandwith of 1600MHz$\times$8bytes = 12.8GB/s per module. With two modules in dual-channel configuration this comes down to 25.6GB/s maximum memory bandwith.

Real bandwith is measured at 18 GB/s for a single core SSE test, and 20 GB/s for a multicore SSE test with ramspeed and ramsmp respectively. Ramsmp can hit the absolute limit of the ram (25.6 GB/s) using pure copy operations. The reason for this difference between single and multicore is not quite clear. The CPU should be able to issue many more instructions than neccesary to hit the bandwith limit, my theory would be that it has to do with how RAM and the CPU actual connect to each other in some way.

2 Optimizations

2.1 Color conversion

2.1.1 Cache

First, we change the loop so that each array is only accessed sequentially. This makes sure we use a whole cacheline everytime we read one before it is thrown out. This is really all there is to do. We read, we compute, than we write. The profiler confirms this, but it is not really needed for these kinds of optimizations...

2.1.2 SIMD

We can change the floating point computation to be a a series of vectorized FMAs. This is perfect usage of the SIMD units, but we are not compute bound so the effect is minimal (we loop fewer times so the overhead is slightlly less for example, and we issue less instructions). However, we only ever write to the output arrays so we can use non-temporal stores to skip having to read them into our cache. As these are only considered hints to the processor, actual performance remains effects remain to be seen. Again the profiler confirms that we are not bound by computation but mostly by memory access. We get a slightly lower "CLK_UNHALTED" count (meaning the processor is doing less work) but the cycles spend waiting on memory is increased by the same amount.

If we use only SIMD and not the cache optimizations we get a performance increase because we waste less bytes put into our L1.

2.1.3 OpenMP

For the non cache-optimized version we can use the dynamic scheduler to hopefully make the cores use the shared cache instead of just system memory. For the optimized version the static scheduler performs better.

2.2 Downsample

2.2.1 Cache

Again the loop is the wrong way around, first we flip that. Then it is also the case that we proccess two channel seperately, we change the code to fuse the two loops and handle both channels at once. After that little remains to be done, as this is a simple copy operation.

2.2.2 SIMD

There is no computation whatsoever going on here, so out uses for SIMD are limited. However, we can (and will) load an entire cacheline using two SIMD load instructions, and then use permutation instructions and blending instruction to efficiently generate 8 outputs at once and then store those in one go. We can also once again use them for non-temporal stores, but only in the optimized case otherwise the store buffer will fill before we fill a cache-line with stores so it will have to fetch from ram before being able to evict anyway.

2.2.3 OpenMP

same as before.

2.3 Concerning data movement

The lab said to look at Color conversion and downsampling. There is (atleast) a function in between to perform lowpass filtering. This function is well suited for blocking optimizations as it accesses its neighboring pixels for each pixel. However, since that is beyond the scope suffice to say that the effect of data movement is that the images are too large to store in even the L3 cache and thus will always be read from memory for each individual step. If we were to edit the entire sequence and remove output functionality, we could perform color conversion, lowpassing and downsampling in a single step requiring us to load much less data and most likely execute and the FLOPS limit of the processor.

2.4 Concerning compiler optimizations

The compiler does not do a very good job at optimizing data access or inserting SIMD instructions. This is partly the fault of the language as it does not give enough restrictions for the compiler to assume a lot of things like alignment. Especially when one needs to make large changes like data-blocking or more complicated SIMD instructions (see for example the SIMD version of the downsample function) the compiler has almost zero chance of doing these optimizations automagically. It is however very good at making sure that once you have specified what you want to do with SIMD instructions to order them in such a way as to maximize the pipeline friendliness of the code and doing register allocation. Although in some cases it is useful to use explicit ymm0 through ymm15 variables.

3 estimates

3.1 Color conversion

Optimizations	**speedup**	**explaniation**
SIMD	8×	The large stride causes us to miss in the cache every iteration. While we are not compute limited, using SIMD to load 8 elements at a time means we should waste only 8 out of 16 instead of 15 out of 16 floats loaded. Experience has shown though that the actual performance increase is usually lower when we are memory bound.
Cache	up to 16×, probably close to this	by changing the loop order around we now access consuctive memory adresses. Baring any evections, which are unlikely on an 8-way cache, this means we should utilize our memory bandwith 16 times better. We are not compute bound even for scalar computations because we have to fetch from RAM as the image is too large large for the cache.
SIMD & Cache	16×	Same reasons as above. same reason as above, maybe even slightlly more with non-temporal stores.
With OpenMP		
Baseline	≈ 3×	As long as the threads run their inner loop somewhat in lockstep they will have shared cache-lines to read from instead of memory. OpenMP gives me 4 threads, but let's be conservative, and thereforeestimate 3 times the gain.
SIMD	50%-100%	For similar reasons we should hit in the shared cache-lines more often now.
Cache	10%-25%	Based upon the benchmarks for RAM bandwith.
Cache + SIMD	10%-25%	Based upon ram benchmarks.

3.2 Color conversion

Optimizations	**speedup**	**explaniation**
SIMD	8×	After optimizations we always read one cacheline per 8 output, as opposed to a cacheline per output.
Cache	8×	Same reason as above. We should use all the data we read only once.
SIMD & Cache	8×	Same reason as above, maybe even slightlly more with non-temporal stores. In any case we will be memory limited.
With OpenMP		
Baseline	$3-4\times$	Again we have 4 cores, using the dynamic scheduler they should hit in the shared cache quite often.
SIMD	30%-40%	We are memory bound and we will not benefit from shared cachelines because each iteration processes one cachline, save for the writes. Add the increased bandwith and the reduction in used bandwith (we lose one write operation out of 6 memory operations).
Cache	10%-25%	Based upon ram benchmarks.
Cache + SIMD	10%-25%	Based upon ram benchmarks.

4 Achieved

4.1 Color conversion

Optimizations	**avg. runtime (ns)**	**speedup**
Baseline	1.3e8	1
SIMD	2.2e7	5.2
Cache	1.3e7	10
SIMD & Cache	1.1e7	12
With OpenMP		
Baseline	3.7e7	3.5
SIMD	1.5e7	8.7
Cache	5e6 - 9e6	26
Cache + SIMD	5e6 - 9e6	26

4.2 Downsampling

Optimizations	**avg. runtime (ns)**	**speedup**
Baseline	1.6e7	1
SIMD	3.3e6	4.8
Cache	1.9e6	8.4
SIMD & Cache	1.2e6	13
With OpenMP		
Baseline	4e6	4
SIMD	1.2e6	8.7
Cache	1.1e6	14
Cache + SIMD	1.1e6	14

5 Discussion

I think this[1] blog-post sums up the results quite well. It is increblly hard to hit full peak bandwith and the 'rules' one has to play by are very arcane, even more so than general optimization on Intel processors. In most cases our estimates where pretty on target, looking at the profiler it seems we were blindsided by the last level cache when estimating our performance (it hid some memory accesses from us) but otherwise we seem to be memory bound in most cases. We also forgot to take into account that in the base cases, writes would first be read from the memory into cache, modified, and then put back into higher level cache. Because we have such a large L3 and L4 cache this means that not all writes will have to touch memory. Many will in fact stil be cached. This gives us a slightlly better performance than originally anticipated accounting mostly for the discrepancies in our predictions. Using the availible hardware counters our predictions hold up for the most part.

However, we I do no know how to explain the fact that the optimized color conversion is more than twice as fast when multithreaded even though it is memory bound. Especially given that the downsampling functions do not improve significantly. We have triple-checked using Intel's vTune profililer however that the runtime is almost soley waiting for memory accesses by taking the cycle count spend waiting on memory (a hardware perfomance counter) and multiply it by the time-per-cycle.

If we were to optimize the whole pipeline we should be able to greatlly increase the performance still. As this has shown, even with a huge L4 cache we keep having to hit memory with this "step-based" design. If we were to optimize the function "lowPass" which sits in between the color conversion and downsampling as well we could merge all three for an I frame, in which case we would only read the data once to perform all three steps. The downsampling is well suited for optimizations anyway as it use neighbouring pixels in both dimensions, this means cache (and register) blocking could give us a great improvement.

[1]http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html

Lastlly, because we are so heavilly memory bound, it might also be beneficial to store the initial image in a byte array and convert it to floats on the fly, this requires only two (cheap) isntructions to do and would reduce the bandwith requirement by 4. Intermediate storage could also be done in either bytes or half-floats.

6 Comments

It is not very practical to have to use the lab computers & Windows. ”luckily” I did this lab by myself and I have a Haswell laptop so I could run it on my laptop but the one time I was working in the lab I got kicked out after an hour because it was reserved for another class. I wasn’t even using the pc then... It may have been how I approached it as well, but it seems educationally a bit weird that all cases are memory bound.