Mincheol Sung

2.1.1.2
vectorized version runs 2 faster than naïve version. Naïve version costs 2.031 seconds, in contrast, vectorized version costs 1.099 seconds. It demonstrates that with SIMD, we can use a data parallelism, and it increases the speed.

2.1.1.3
(1) __m128 _mm_setzero_ps (void) : Return vector of type __m128 with all elements set to zero.

(2) __m128 _mm_loadu_ps (float const* mem_addr) : Load 128-bits (composed of 4 packed single-precision (32-bit) floating-point elements) from memory into dst. mem_addr does not need to be aligned on any particular boundary.

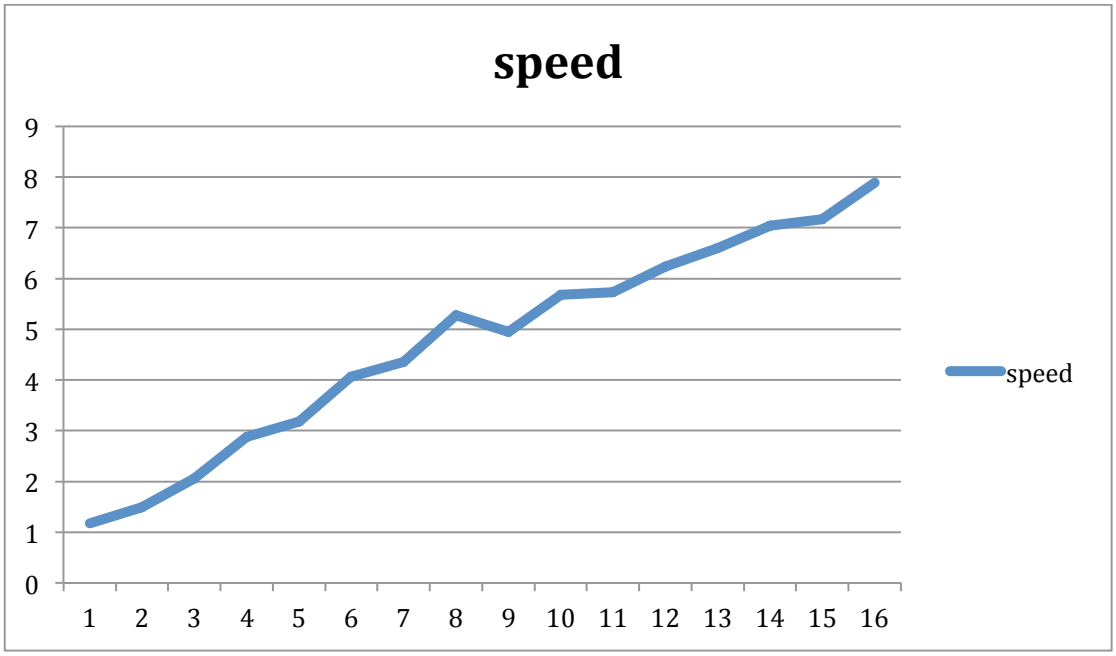
(3) __m128 _mm_add_ps (__m128 a, __m128 b) : Add packed single-precision (32-bit) floating-point elements in a and b, and store the results in dst.

(4) void _mm_storeu_ps (float* mem_addr, __m128 a) : Store 128-bits (composed of 4 packed single-precision (32-bit) floating-point elements) from a into memory. mem_addr does not need to be aligned on any particular boundary.

2.1.1.4
I just follow naïve version's kernel.
With load instruction(_mm_loadu_ps) I load data from frame[]. Then, with add instruction(_mm_add_ps) I add vector_a and temp and put it into temp.
Out of the loop, with store instruction (_mm_storeu_ps), I store data of temp in avg_array[]. Finally, I get avg from avg_array[] and calculate the average with "avg/num".

2.1.1.5
I considered both unaligned and aligned loads. Unaligned load instruction is _mm_storeu_ps, and aligned load instruction is _mm_store_ps.
I expected that aligned load would be more faster, because of use of cache, so I tried. However, I couldn't find correct way of aligned load, I just made my own; aligned_vector_blur.cpp. In my code, because there are more memory access then unaligned version, It costs more time. It costs about 1.6 seconds. My concept for aligned load is that the load instruction is called with index of frame[] starting at 0. To align it, I have to put a pad. The pad can be null to just occupy memory.

2.1.2.2



2.2.1.3
The kernel is almost same as the naïve version.
I just put #pragma omp parallel for private(c). One of the best merits of Openmp is handy. I can make parallel program very easily with just one sentence. To make variable C independent with threads, I use “private” clause.

2.2.1.4
I considered load balancing. I can do it with Openmp schedule.
With “#pragma omp parallel for private(c) schedule(dynamic)”, it balance the load of each threads, and by default it’s chunk size is 1.
It is 10.31 faster than the naïve version. It is faster than the parallel version without load balancing.

2.3.2
The fastest version is 11.2 faster than the naïve version.

2.3.3
I make it with SIMD and multi-thread together. With Openmp and “#pragma omp parallel for private(c) schedule(dynamic)” with load balancing, and in the loop, I use SSE instructions which I used in vertorized_blur. However, the result is less that what I expected.