Sriharsha Ayinala

1378833

2.1.1

1) Refer to source code.

2) The code ran 2.18x faster. The serial algorithm took 2.349s while the SIMD implemented calculation took 1.077s.

3)
_mm_loadu_ps – This is used to load unaligned single precision floating-point data from memory using the frame pointer into the SIMD registers.
_mm_hadd_ps – This is used to do a horizontal add between two SIMD registers. This enables 4 additions simultaneously.
_mm_store_ps – This is used to store the calculated data after addition in the SIMD registers back into memory.

4) The code first initiates the vectorized loop where it calculates the sum of a 4x4 block starting from the top left corner of the blur block. It will try to fit this much 4x4 blocks into the loop to do the calculations. The fringe rows , or the remainder rows that are not divisible by 4, are then added using lesser vectors. The amount of vectors used is determined by the remainder. The fringe columns, are then calculated serially. All these values are added up into the average divided by the size of the blur block.

5) Aligned/unaligned loads were considered. Because the frame values in memory might not be aligned, I used the load unaligned intrinsic instruction to load the values properly. Load unaligned decreases performance. To get better performance, I would not use this intrinsic instruction but instead grab the aligned values in the middle and deal with the unaligned values serially.

Sriharsha Ayinala

1378833

2.2.1

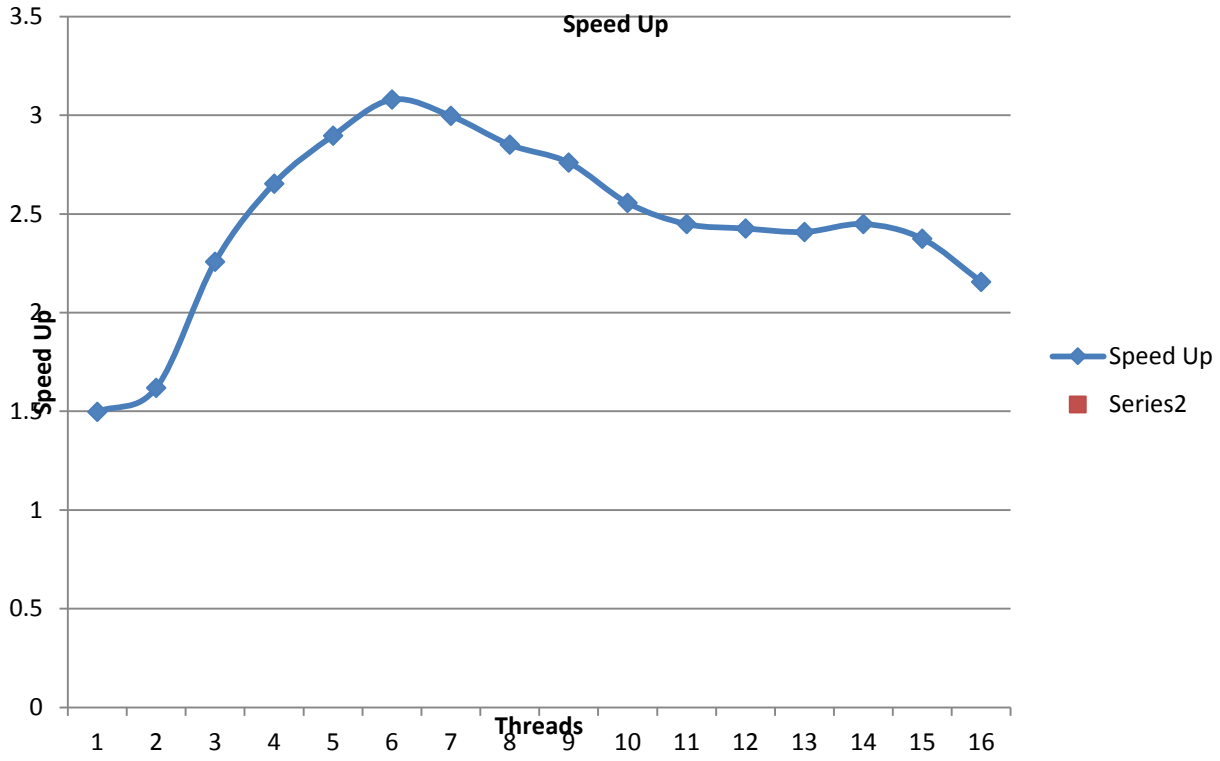
1) Refer to source code.

2)

Threads	Time	Speed up
Serial	2.346	-
1	1.566	1.498084
2	1.449	1.619048
3	1.039	2.25794
4	0.884	2.653846
5	0.81	2.896296
6	0.762	3.07874
7	0.783	2.996169
8	0.823	2.850547
9	0.85	2.76
10	0.918	2.555556
11	0.958	2.448852
12	0.967	2.42606
13	0.974	2.408624
14	0.958	2.448852
15	0.988	2.374494
16	1.088	2.15625

Sriharsha Ayinala

1378833



3) The program parallelizes the for loop that iterates through the rows and columns of the frame. The work that is done inside the for loop is the blurring of a pixel in the frame. The loop is broken up into rows of tasks that is each assigned to a specific thread to compute based on the ID.

4) Load-balancing was not considered. This is probably why after 5 threads does not improve. It is bottlenecked by the slowest thread to complete. To implement load-balancing, I would find the middle ground for computation for each thread. The tasks for each thread would be divided up some into sufficiently smaller tasks such that overhead is not the biggest portion of the work by the same time not too large that it causes load imbalance. One such way to implement load-balancing is to assign computation of multiple rows of output pixel as a single parallel task. Depending on the size of these rows, the size of these computational task would vary for the load-balancing scheme.

2.3

1) Refer to source code.

Sriharsha Ayinala

1378833

2) The new algorithm is now 21.54x faster than the naïve algorithm. The serial time taken is 2.348s while the fastest time is 0.109s with 16 threads.

3) I implemented load-balancing by splitting the tasks needed to be done into smaller groups rows. I then used OpenMP's task and distributed the individual tasks into available threads. The remaining fringe rows are calculated serially.