1 Vectorization

1.1 Code

Listing 1: vector_blur.cpp

```
#include "my_blurs.h"

using namespace std;

void vector_blur(float* out, int n, float* frame, int* radii){
  for(int r=0; r<n; r++)
    for(int c=0; c<n; c++){
      int rd = radii[r*n+c];
      int num = 0;
      float avg = 0;
      __m128 avg4 = _mm_set1_ps(0);
      for(int r2=max(0,r-rd); r2<=min(n-1, r+rd); r2++) {
        for(int c2=max(0, c-rd); c2<=min(n-1, c+rd);){
          if (c2 + 4 <= min(n-1, c+rd)) {
            __m128 vals = _mm_loadu_ps(frame + r2*n + c2);
            avg4 = _mm_add_ps(avg4, vals);
            num += 4;
            c2 += 4;
          } else {
            avg += frame[r2*n+c2];
            num += 1;
            c2 += 1;
          }
        }
      }
      float avg_a[4];
      _mm_storeu_ps(avg_a, avg4);
      for (int i = 0; i < 4; i++)
        avg += avg_a[i];
      out[r*n+c] = avg/num;
    }
}
```

1.2 How Much Faster

The vectorised code ran 1.54 times faster than the naive blur. Ideally, we should get up to 4 times faster. However, the reduced performance it is probably due to two factors. Firstly, the ratio of edge cases (where we cannot properly use SIMD instructions) to good cases (where we can use the SIMD efficiently) is relatively high, and this is true for small blurring radiuses and resolution. Secondly, we are adding some floating point adds to accumulate the values of the vectors.

1.3 Kernel Organisation

Let $p_{r,c}$ be a pixel in position (r, c) in the blurring frame, and define the SIMD vector $\mathbf{V} = \langle v_0, v_1, v_2, v_4 \rangle$ with

$$v_t = \sum_{i,j \equiv t \bmod 4} p_{i,j}$$

So, for each row of the pictures we are accumulating all the pixels with column value $\equiv t \bmod 4$, in the frame, to v_t. Then, once we reach the end of the pixels in the frame we accumulate the four values to the sum used for the blurring.

1.4 Aligned/Unaligned Loads

Intel states that "If alignment cannot be guaranteed, some part of the performance gain achieved by processing multiple data elements in parallel will be lost because either the compiler or assembly programmer must use unaligned move instructions," which for most architectures are slower than aligned move instruction. In this implementation, aligned instructions were not utilised due to simplicity. One approach would be to use the vectorization on aligned memory blocks of 4 floats within the blurring frame and then deal with the edge cases using the unvectorized approach to the algorithm. Using Intel's reference on SIMD, this implementation should have a speedup of something below 2x on Core i7 CPU.

2 Parallelization

2.1 Code

Listing 2: parallel_blur.cpp

```
#include "my_blurs.h"

using namespace std;

void parallel_blur(float* out, int n, float* frame, int* radii){
#pragma omp parallel for schedule(dynamic)
  for(int r=0; r<n; r++)
    for(int c=0; c<n; c++){
      int rd = radii[r*n+c];
      int num = 0;
      float avg = 0;
      for(int r2=max(0,r-rd); r2<=min(n-1, r+rd); r2++)

        for(int c2=max(0, c-rd); c2<=min(n-1, c+rd); c2++){
          avg += frame[r2*n+c2];
          num++;
        }
      out[r*n+c] = avg/num;
    }
}
```

2.2 Scaling Plot

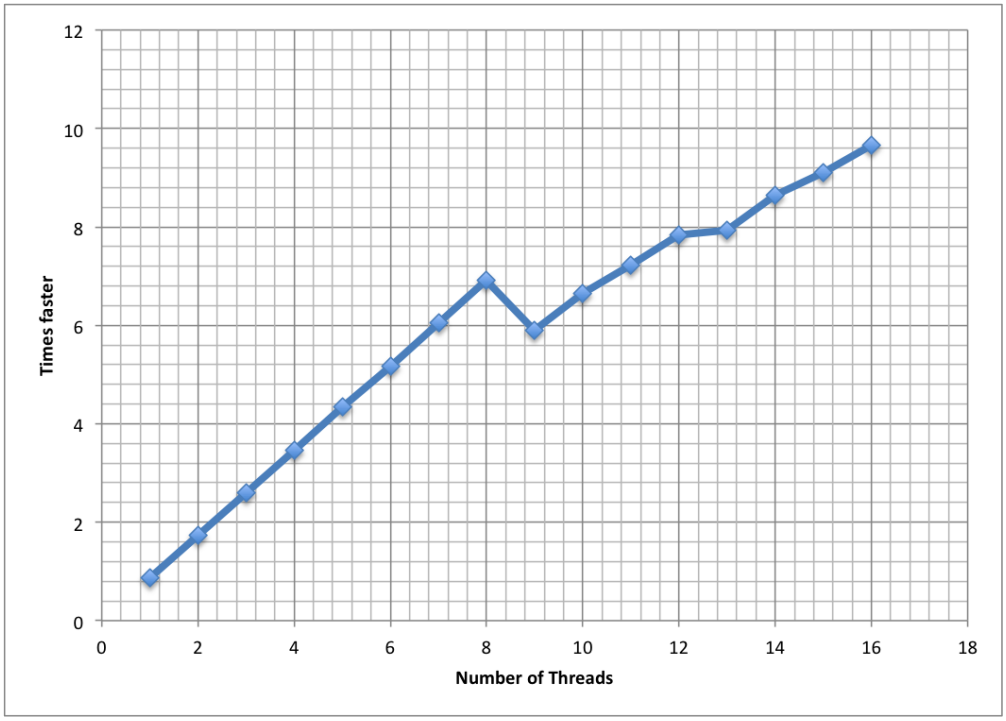

Figure 1: Scaling Plot

2.3 Kernel Organisation

We simply assign every row of the picture frame to a different thread using parallel for flag on the outer loop of the algorithm.

2.4 Load Balancing

Load balancing was approached using the *schedule* flag with *dynamic* option on parallel for flag. Threads that are closer to the top and bottom frames will do less work than threads in the centre of the picture frames. The dynamic option will balance the load at run time.

3 Fastest Implementation

3.1 Code

Listing 3: fastest_blur.cpp

```
#include "my_blurs.h"

using namespace std;

void fastest_blur(float* out, int n, float* frame, int* radii){
#pragma omp parallel for schedule(dynamic)
```

```
  for(int r=0; r<n; r++)
    for(int c=0; c<n; c++){
      int rd = radii[r*n+c];
      int num = (max(0,r-rd) - min(n-1, r+rd) - 1) * (max(0, c-rd) - min(n
         -1, c+rd) - 1);
      float avg = 0;
      __m128 avg4 = _mm_set1_ps(0);
      for(int r2=max(0,r-rd); r2<=min(n-1, r+rd); r2++) {
        for(int c2=max(0, c-rd); c2<=min(n-1, c+rd);){
          if (c2 + 4 <= min(n-1, c+rd)) {
            __m128 vals = _mm_loadu_ps(frame + r2*n + c2);
            avg4 = _mm_add_ps(avg4, vals);
            c2 += 4;
          } else {
            avg += frame[r2*n+c2];
            c2 += 1;
          }
        }
      }
      float avg_a[4];
      _mm_storeu_ps(avg_a, avg4);
      for (int i = 0; i < 4; i++)
        avg += avg_a[i];
      out[r*n+c] = avg/num;
    }
}
```

3.2 How Much Faster

The fastest implementation is 11.6 times faster than the naive blur.

3.3 Configuration

We simply assign every row of the picture frame to a different thread using parallel for flag on the outer loop of the vectorised algorithm.