CS194-15 Assignment 3: Optimizing Image Blur

Created By: Patrick Li, David Sheffield

Assigned: September 29, 2014

Due: October 6th 2014, 11:59 pm

Logistics

- Hand in a report containing your answers to all of the questions outlined in the *Task* sections. We expect roughly 1-2 sentences for 2 point questions. 2-3 sentences for 3 point questions. 3-4 sentences for 4 point questions. Questions requiring coding will be worth more points.
- There is a 10% grade penalty for each day late, up to a maximum of 7 days, after which we will no longer accept your assignment.
- Each student is allowed 5 days slack for the year which may be used for postponing handing in an assignment with no penalties. However, a maximum of 7 days late still applies to every assignment.
- At the end of the semester, any slack days unused can be redeemed for a bonus 10% grade on any assignment.

1 Image Blur Kernel

This assignment asks you to optimize a simplified version of the image blur kernel introduced in the second assignment. Unlike in assignment 2, the blur now works on only square grayscale images instead of rectangular three-channel images. The included file "blur.cpp" contains a naive implementation of blur and test code for checking whether your blur implementation returns the same result as the naive implementation, and the time it takes to run each of them on a square 3000×3000 sized image.

The provided code can be compiled with the command:

```
g++ -msse -O2 blur.cpp -o blur
```

and run with the command:

```
./blur
```

1.1 Explanation of Kernel

```
void simple_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;
      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;
    }
}
```

The above naive implementation takes four arguments:

- float* out: A $n \times n$ sized array of floats where the output of the blur will be stored.
- int n: The width (and height) of the image to be blurred.
- float* frame: A $n \times n$ sized array of floats representing the frame to be blurred stored in row-major order.
- int* radii: A $n \times n$ sized array of ints representing the radii to be used for blurring at each pixel, also stored in row-major order.

The outer two loops iterate over each pixel in the frame. *rd* contains the radii to use to compute the blur for the current pixel (r, c). *num* keeps track of the number of pixels in the neighbourhood of the current pixel. For pixels away from the border of the image, *num* will be equal to $(2 \times \text{rd} + 1)^2$, but *num* will be smaller for pixels along the borders of the image. The inner two loops iterate over each pixel in the neighbourhood of the pixel (r, c) with r_2 lying within $[r - \text{rd}, r + \text{rd}]$ and c_2 lying within $[c - \text{rd}, c + \text{rd}]$. The *min* and *max* operations ensure that the neighbourhood does not extend past the boundaries of the image. Within the two inner loops we accumulate the sum of the pixel values in *avg* and accumulate the number of pixels in the neighbourhood in *num.* Finally the computed blurred pixel value (avg/num) is stored in the output array *out.*

2 Your Job

Your assignment is to optimize the image blur kernel above to run as fast as possible. You may use whatever techniques you want, but there are two that will be required, vectorization and parallelization. Space is provided within *blur.cpp* for you to insert your implementations.

2.1 Vectorization

Intel and AMD processors come with a special set of instructions, called SSE instructions, for directly controlling the simd functional units. SSE intrinsics are a pleasant C wrapper for these instructions, so that the user does not need to write assembly code to gain access to this functionality. Using SSE intrinsics you will be able to perform four additions simultaneously! Using the information in the "Intel Intrinsics Guide" take advantage of the SSE instructions to vectorize the image kernel.

2.1.1 For Your Report

1. Submit your code as *vector_ blur.cpp.* (40 points)
2. How many times faster did your vectorized version run? (4 points)
3. What intrinsic instructions did you use? (4 points)
4. Describe how you organized your kernel using those intrinsic instructions. (4 points)
5. Did you consider the unaligned/aligned loads? How did/would you address that? (4 points)

2.2 Parallelization

Using the parallelization framework of your choice (OpenMP or PThreads), take advantage of the multiple cores available on your machine to parallelize the vectorized version of your blur kernel from section 2.1.

2.2.1 For Your Report

1. Submit your code as *parallel_ blur.cpp.* (40 points)
2. Create a scaling plot with the number of times faster than the naive version on the vertical axis, and the number of threads used on the horizontal axis. Vary the number of threads from 1 to 16. (16 points)
3. Describe the organization of your parallel kernel. (4 points)
4. Did you consider load balancing? How did/would you address that? (4 points)

2.3 Fastest Implementation

1. Submit your fastest configuration as *fastest_ blur.cpp.* (10 points)
2. How much faster is it compared to the naive implementation? (2 points)
3. Describe the configuration you used. (4 points)

2.4 Bonus: Algorithmic Improvements

There are two critical concerns that must be addressed to optimize any piece of code. The first concern is whether the code is executing more instructions than necessary to compute the result. The second concern is whether your machine is executing as many instructions as possible per clock cycle. The previous questions have been focusing on the latter concern but this question focuses on the former.

The naive implementation of the blur given to you performs *many* more additions than is necessary. Devise an algorithm that performs substantially less additions to compute the same result. A hint is to look up *2D cumulative sum* online.

2.4.1 For Your Report

1. Describe how to improve the algorithm. (4 points)
2. Implement your efficient algorithm and submit it as *better_blur.cpp*. (15 points)
3. How much faster is it than the naive implementation? (2 points)