**PARALLELIZING IMAGE FILTERING TECHNIQUES**

Kevin Gregor – krg43, Jaydev Kshirsagar – jlk368

**ABSTRACT**

The process of image filtering can be inherently parallelized since each operation that happens on any given pixel is completely independent of operations on other pixels. Hence, we have a lot of scope for exploiting the parallel nature of the problem. For large image sizes, applying an image filter serially will take a significant amount of time. We explore multiple methods of performance tuning, including memory layout adjustments and parallelization using OpenMP. We primarily attempt to parallelize the Finite-Impulse Response (FIR) type of filters that operate in the spatial domain. We have implemented a generic framework than can apply any FIR filters with a 3x3 kernel. Configuring the kernel appropriately can help achieve the desired type of effect.

1. **BACKGROUND & RELATED WORK**

There are a variety of different image filtering techniques, but we focus specifically on convolution filtering in the spatial domain. This means that we are performing a transformation on the value (i.e. the color) of a pixel, where we change the value to a weighted average of the pixel itself and all the neighboring pixels. In order to do this, we use a 3x3 matrix, called a kernel or mask, to specify the weights of the nine pixels. The center of the matrix denotes the pixel itself and the other eight locations in the matrix are the neighboring eight pixels. Below is an example of the kernel for a Gaussian Blur, a low-pass type filter.

|  |  |  |
| --- | --- | --- |
| 1/16 | 1/8 | 1/16 |
| 1/8 | 1/4 | 1/8 |
| 1/16 | 1/8 | 1/16 |

Fig 1. Gaussian Blur Kernel

Although neither of us had extensive experience in the area of image processing, we did have a preliminary understanding of signal processing concepts. A plethora of work has been done in the image processing field, and we were able to quickly learn about exactly what image filtering is and how it is implemented using various resources online.

We also found a few studies on the speedup resulting from parallelization of image filtering. As mentioned in the abstract, convolution filtering is a great candidate for parallelization because all of the pixel computations are independent of each other. In one study we found, which was done by students at Purdue University, the students tested parallelization using both OpenMP and MPI. They found that increasing the number of processes using OpenMP led to more of a speedup than doing so with MPI. Below is a graph of their findings:

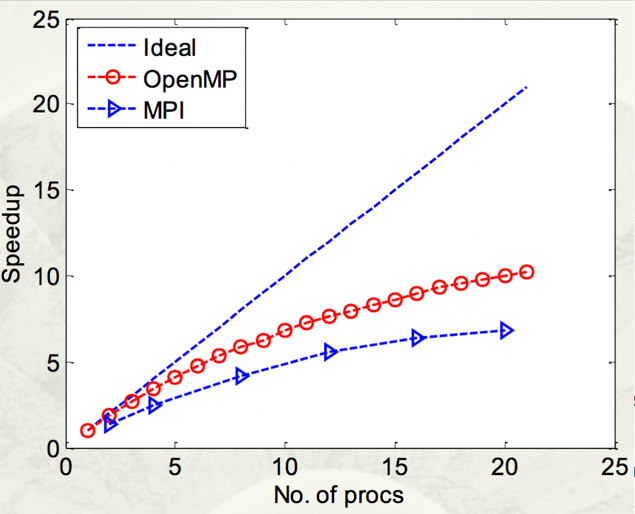


Fig 2. Speedup of OpenMP and MPI [1]

Another study by Devrim Akgün of Duzce University in Duzce, Turkey, also got similar results for the impact of number of threads/processes on speed up for image processing. From his graph below, he got a speedup of 5-6 times using six threads – similar to the Purdue students’ speed up of just under 5 times using six processes.

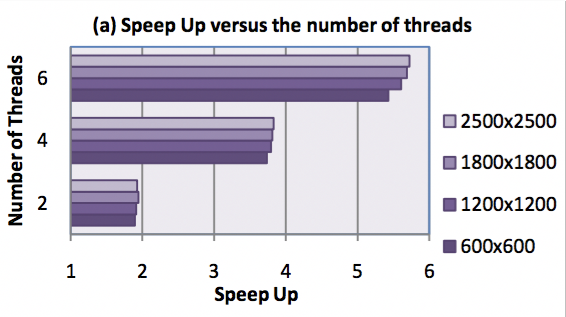


Fig 3. Speed up vs Number of threads [2]

As demonstrated by these studies, there is a fair amount of speedup resulting from simple parallelization, which we aimed to replicate. We were not able to find information regarding the impact of memory layout adjustments on speed up, so we were unsure of exactly how our changes would impact performance.

1. **IMPLEMENTATION DETAILS**

The core implementation of our project, which includes everything except for reading the images and running through test cases, was written in C++. We created a shared C++ library, which is then interfaced with by a Python script. Our Python testing script reads an image into a NumPy matrix and creates a C++ object for the image based on the matrix, its dimensions, the image format, and the desired memory layout format. Then, after the filter is applied using C++, the processed image matrix is passed back into the Python script and written to the output image file. We chose to abstract the image reading and writing to a Python script because implementing it in Python is quick and easy due to the availability of libraries and also because this was not a performance critical part of the project. This also allowed us to run through test cases quickly using the Python script.

The C++ code when invoked through Python, creates an object for the image that stores all essential attributes of the image,  
and at the same time allocates required amount of memory for storing the pre-processed and the processed image data. Also the input pixel data in one of the supported image formats, is read and the pixel values are normalized to [0, 1], and the pixels are stored in the specified memory layout.

When the Python script calls in to the library for getting the image processed, the specified filter kernel is convolved with the image data and the resultant image data is stored in the memory area allocated for the processed image. The pixel values of the processed image data are still in the normalized form and in the specified layout.

When the Python script requests back the processed image data, the library code converts the pixel values back to the desired format and also converts the image layout back to the natural one; the converted image data is written to a pointer supplied by the Python script. The Python script subsequently writes the image data to a file.

The supported image formats or pixel data formats are 8-, 16-, and 32-bit grayscale images, 24-bit RGB images with byte-alignment of 3 and 4 for the pixels. The image format has to be specified when using this application.

The parallelization of the computation involved in the processing, is not very difficult and also the performance is not limited by compute, but by memory bandwidth. Hence, it was more important to use optimal memory layouts for the images so as to maximize the cache efficiency. We implemented two special memory layouts that would be suitable for the spatial locality of the pixel references that are involved in the filtering computation. We also implement a naïve row-major layout for baselining purposes. This implementation is not ideal for applying 3x3 matrices, as there is not a strong sense of data locality – only three of the nine pixels needed for each computation are in the same row as the pixel we are doing the computation on. Clearly, this can be improved upon by using another memory layout.

The first special layout that we implemented (which is actually having 3 variants) is the Block-Linear layout with block sizes of 8, 16, and 32. This involves dividing the image into blocks of fixed sizes (*n*-by-*n* (8/16/32)). The blocks stored in row-major order, and the pixels within each block are also stored in row-major order. Therefore, the data locality can be greatly increased, as each pixel needed for computation will be no more than indices away from the pixel being modified. For image sizes that are not perfect multiples of the block size, we add extra “dummy” pixels to fill up the remaining rows and/or columns in order to create a matrix that can be blocked perfectly. These dummy pixels are not touched when actually applying the filter and they are not saved as a part of the image after the filter has been applied.

The second special memory layout adjustment we implemented was a twiddled Z-Morton style layout. Our Z-Morton layout can be seen below in the figure below.

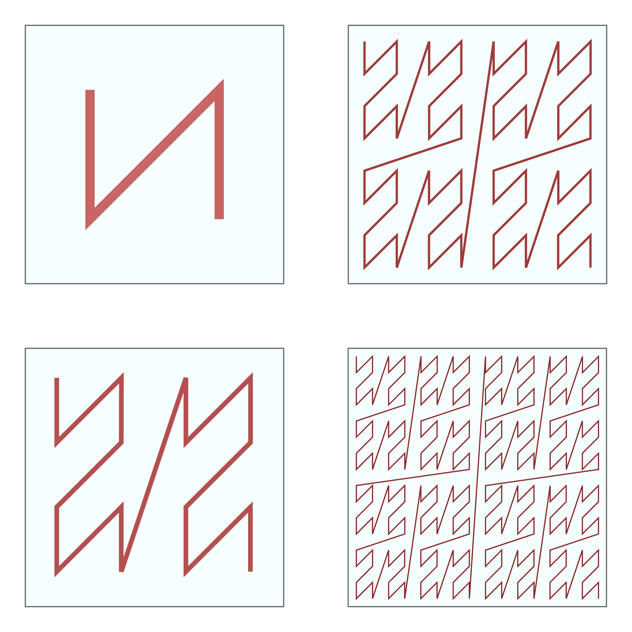


Fig. 4. Z-Morton layout visualization

We divide the input matrix into four large blocks – top left, bottom left, top right, bottom right – that are stored in memory in that order. Similar to the implementation of the Block-Linear layout, we recursively divide the blocks into smaller and smaller blocks, applying the same ordering technique, as seen in the Figure 4. In theory, this gives us even better data locality than the Block-Linear layout, due to pixels in any given 3x3 grid being stored very close together in memory. Moreover this layout is better suited for hierarchical caches, since the data corresponding to the higher level of the recursion tree lies in the next higher level of cache. Just like with the Block-Linear layout, this layout also suffers when image dimensions are not perfectly aligned. However, the Z-Morton layout needs to add dummy pixels when the image size is not a power of two, and dummy pixels are added to round up to the next power of two.

Both of our non-naïve memory layouts also have a hidden cost of converting the input matrix into the desired layout before applying a filter and back afterwards. Since FIR filters can already be applied very quickly on an image of any reasonable size, converting to and from these layouts just to apply one filter proves not to be worth it: the total time from reading the input image to writing the output image is longer than the naïve version. However, we hope that the speedup in the actual filter application means that using the Block-Linear or Z-Morton layouts would be better when applying multiple filters in succession.

1. **PERFORMANCE ANALYSIS**

For performance analysis all three memory layouts for the input image are tested quickly in succession. Also all three types of filters are applied.

* 1. **PEAK PERFORMANCE**
  2. **TEST CASES**
  3. **BASELINE PERFORMANCE**
  4. **MEMORY LAYOUT CHANGES**

1. **CONCLUSION**

**REFERENCES**

[1] Parallel Image Processing – Lei Cao and Yan Wang (2013-04-19)

https://engineering.purdue.edu/~eigenman/ECE563/ProjectPresentations/ParallelImageProcessingCao\_YanWang.pdf

[2] Performance Evaluations for Parallel Image Filter on Multi-Core Computer using Java Threads – Devrim Akgün (July 2013)

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.403.1706&rep=rep1&type=pdf