**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 special 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**
2. **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 C++ proved to be significantly more complex and time-consuming. This also allowed us to run through test cases quickly using the Python script.

We support a variety of image formats, including 8-, 16-, and 32-bit grayscale images and 24-bit RGB images. The image format can be specified when reading the image in Python, which allows multiple formats for the same image to be tested quickly in succession. We have applied filters to all image formats, and our performance analysis in section 3 will detail the time it takes to apply the same filter on various formats of the same image.

In addition to testing multiple image formats, we also implemented three different memory layouts to see which layout would give us the best performance. The first layout we implemented is a naïve row-major layout of the input matrix. This implementation is not ideal for applying 3x3 matrices, as there is not a strong sense of data locality – only three of the 9 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.

Another layout adjustment that we implemented (which is actually three layouts) is the Block-Linear layout with block sizes of 8, 16, and 32. For this layout, blocks of size *n*-by-*n* (8/16/32) are 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.

For our final memory layout adjustment, we implemented support for a twiddled Z-Morton style layout. Our Z-Morton layout can be seen below in the figure below.

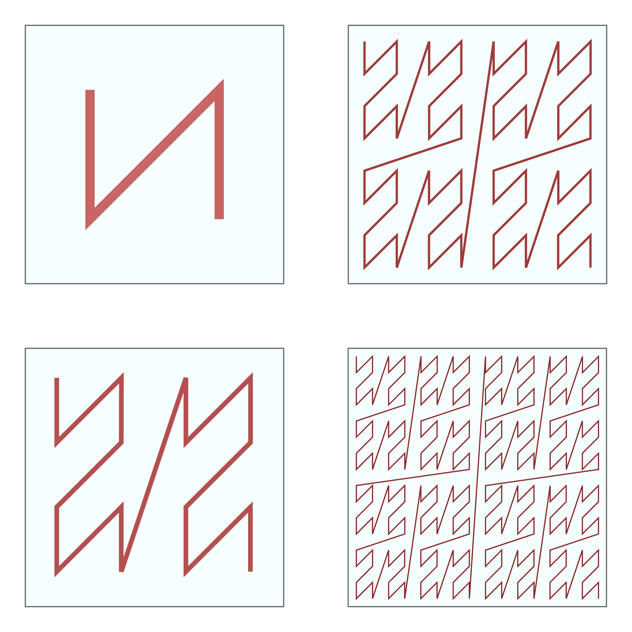


Fig. 3. Z-Morton layout visualization

We divided 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 3. 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. 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** 
   1. **TEST CASES**
   2. **BASELINE PERFORMANCE**
   3. **MEMORY LAYOUT CHANGES**
2. **CONCLUSION**