EEP524
Jaidon Lybbert
12/02/2022

Final Project Design - SIFT

Overview

The goal of this project is to implement the key localization portion of the Scale-Invariant Feature Transform (SIFT) algorithm using CUDA. The input is a grayscale image, and the output is an annotated image marker indicating keypoint location and canonical orientation.

The steps to achieve this are the following:

1. Bilinear Interpolation
2. Gaussian Blur
3. Matrix Subtraction
4. Indexing Difference-of-Gaussian Extrema
5. Gradient Map Generation
6. Keypoint Characterization by Histogram
7. Keypoint Annotation Transformations

Parallel Algorithms

These are the algorithms that make the most sense to implement in parallel, and their corresponding theoretical arithmetic intensities.

Bilinear Interpolation

Implementation

Bilinear interpolation lends itself to a parallel implementation due to the locality of data, and computations needed per pixel. Since the output data is larger than the input data, the number of pixels to compute is more than the number to read from memory. With sufficient spacing, one thread block can load all the required data into shared memory in one read per thread. An existing approach is detailed by Y. Sa[1].

The algorithm to interpolate values takes three steps:

1. Determine the coordinates
2. Interpolate row-wise
3. Interpolate column-wise

The coordinates of the output pixels relative to the input image are calculated through division of the output pixel coordinates by the pixel spacing,

$$x' = x / s$$
$$y' = y / s,$$

where x' indicates the input coordinate, x indicates the output coordinate, and s indicates the spacing.

Row wise interpolation is done to get intermediate values which will be used in the column-wise interpolation. The formula is

$$R1 = Q11 * (x2 - x') / (x2 - x1) + Q21 * (x' - x1) / (x2 - x1),$$

where R1 is the intermediate row-wise interpolated value, Q11 and Q21 are the adjacent pixel values in the same row, and x2 and x1 are the adjacent integer input coordinates.

Column-wise interpolation is then done to get the final result. The formula is

$$R1 * (y2 - y') / (y2 - y1) + R2 * (y' - y1) / (y2 - y1),$$

where R1 and R2 are the previously row-wise interpolated values, and y1 and y2 are the adjacent integer input y coordinates.

A.I.

If the image size is NxM, and the expansion ratio (or spacing) is set at 1.5, then the number of pixel values to compute is Nx1.5xMx1.5. Each pixel value takes approximately 30 arithmetic operations to compute. So, the arithmetic intensity with good data sharing approaches,

A.I. = (Nx1.5xMx1.5x30 / NxM) = 67.5 FLOPs/Byte

Gaussian Blur

Implementation

Gaussian blur is done by convolution of a weighted kernel with the input images. This has been implemented earlier in the quarter, so I am not going to describe it here.

A.I.

If the kernel size is QxQ, then the approximate number of operations per pixel including iterating over the kernel area, adding addresses, and doing the convolution sum and normalization is

QxQx9,

with no data sharing between pixels, this equates to an A.I. of 9 FLOPs/Byte. However, there is opportunity for substantial data sharing. With the 9x9 kernel used in the host application, and a block size of 32x32, using a tiled load, the arithmetic intensity would approach

9x9x9/2 = 364 FLOPs/Byte

Gradient Map Generation

Implementation

The gradient maps are generated pixel-by-pixel and use the neighboring pixel values directly to the right and directly below the current pixel, for a total of 3 input values. In parallel, data can be shared between threads easily, and each thread can compute one value for the gradient magnitude map and gradient orientation map. The equations to compute these values are as given by D. Lowe [2].

$$M_{ij} = \sqrt{(A_{ij} - A_{i+1,j})^2 + (A_{ij} - A_{i,j+1})^2}$$

$$R_{ij} = \operatorname{atan2}(A_{ij} - A_{i+1,j}, A_{i,j+1} - A_{ij})$$

Fig 1. Equations for image gradient magnitude M_{ij} and orientation R_{ij}. Where A_{ij} is the value of a pixel located at coordinate (i, j).

A.I.

If both gradient magnitude and orientation maps are to be computed, then for each pixel of the input, there are more than 50 FLOPs to be computed, based on estimated FLOPs of the square root and arctan functions. So, the AI would approach 50 FLOPs/Byte.

Key Point Characterization

Implementation

The keypoint characterization step requires building a histogram of gaussian weighted local gradient orientations. In parallel, this implies a 2D gaussian kernel stored preferably in shared memory as it will be accessed frequently by all threads. To share memory as much as possible, a large block size compared to the kernel size is desired, and sections of the gradient map, whose values are in the range of 0-360 degrees, will be loaded collaboratively into shared memory.

Each thread will own a histogram which is 36 entries long, corresponding to 360 degrees of rotation with a resolution of 10 degrees. The thread will iterate through the surrounding values in the section of the gradient map stored in shared memory. Each surrounding value will be divided by 10 and floored to an integer value to obtain the histogram index. The corresponding gaussian weighted value will be accumulated in the histogram index.

After iterating through all surrounding values, the peak of the histogram will be determined, and saved in global memory.

A.I.

If the gaussian kernel is QxQ, the approximate FLOPs for one keypoint is QxQx38. With good data sharing, the arithmetic intensity would approach

QxQx38/2,

which, for the 5x5 kernel used in the host implementation, amounts to

5x5x38/2 = 475 FLOPs/Byte

Host Reference Implementation

The host implementation contains functions roughly corresponding to the steps outlined in the overview. An example of the algorithms input and output is shown below. Using the test image, about 4,500 key points were identified for the 240x240 image, which falls in the range I would expect based on the paper by D. Lowe.

Fig 2. Input image to the sequential SIFT key localization algorithm.

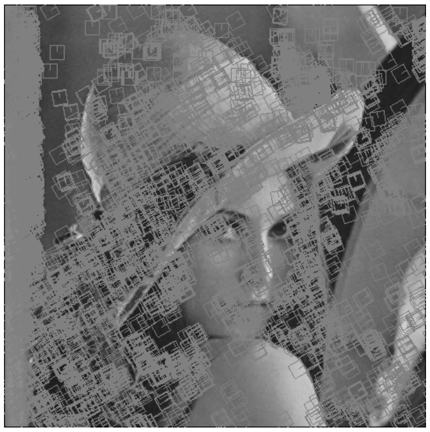


Fig 3. Annotated output from the algorithm, showing 4,500 key locations and orientations.

The number of operations for each step of the algorithm depends on the image size, and how many layers are chosen for the construction of the image pyramid. Approximate complexity and numbers of operations for the portions of the algorithm most suitable for parallelization are the following:

Bilinear Interpolation

If the image size is NxM, and the expansion ratio (or spacing) is set at 1.5, then the number of pixel values to compute is Nx1.5xMx1.5. Each pixel value takes approximately 30 arithmetic operations to compute. The complexity is O(NxM), and the number of operations in the implementation is about,

$$\text{FLOPs} = 30 \times (240\times240\times2\times2 + 480\times480\,(1.5^2 + 1.5^4 + 1.5^6))$$

$$\text{FLOPs} = 135\text{M},$$

For a 4 layer image pyramid.

Gaussian Blur

If the image size is NxM and the kernel size is QxQ, then the approximate number of operations including iterating over the kernel area, adding addresses, and doing the convolution sum and normalization is

NxMxQxQx9.

This is repeated 8 times for increasing image sizes, and a 9x9 kernel in the implementation starting from 480x480 for a total of

FLOPs = 9 x 9 x 2 x 480^2 (1 + 1.5^2 + 1.5^4 + 1.5^6)

FLOPs = 735M

Matrix Subtraction

Each matrix subtraction takes 2xNxM operations for a NxM image. This is done 4 times for increasing image sizes for a total of

FLOPs = 2 x 480^2 (1 + 1.5^2 + 1.5^4 + 1.5^6)

FLOPs = 9M

Gradient Map Generation

If both gradient magnitude and orientation maps are to be computed, then for each pixel of the input, there are more than 50 FLOPs to be computed, based on estimated FLOPs of the square root and arctan functions. This is done 4 times for increasing image sizes for a total of

FLOPs = 50 x 480^2 (1 + 1.5^2 + 1.5^4 + 1.5^6)

FLOPs = 227M

Key Point Characterization

The keypoint characterization step requires building a histogram of gaussian weighted local gradient orientations. If the gaussian kernel is QxQ, the approximate FLOPs for one keypoint is QxQx38. This is done once for 4,000 keypoints, with a kernel of 5x5, for a total of,

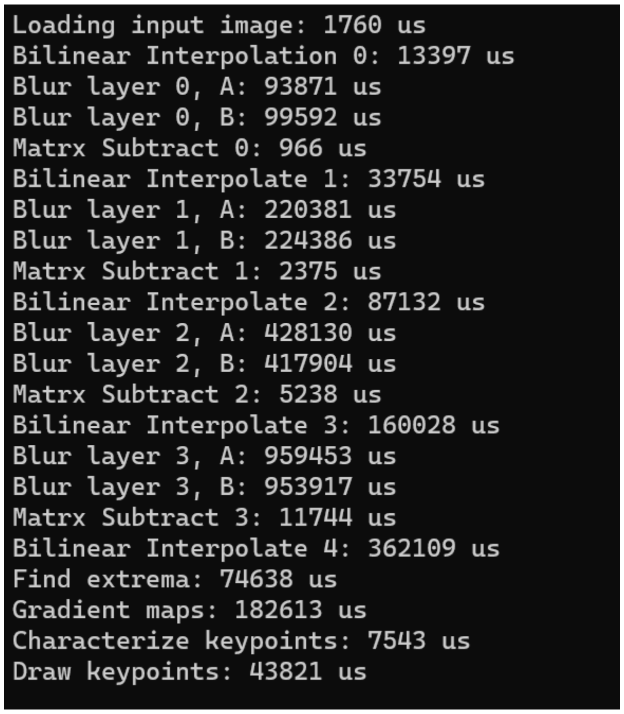
FLOPs = 4,000x5x5x38

FLOPs = 3.8M

Host Reference Timing

Timing the host implementation shows how the algorithms compute times scale with increasing input sizes. The time of execution for the blur, bilinear interpolation, and matrix subtract all show exponential growth at increasing levels of the image pyramid. The input is scaling exponentially as the bilinear interpolation transform grows the input by a factor of 1.5 in each dimension for every layer.

This shows the most time can be saved by parallelizing these three algorithms.



```
Loading input image: 1760 us
Bilinear Interpolation 0: 13397 us
Blur layer 0, A: 93871 us
Blur layer 0, B: 99592 us
Matrx Subtract 0: 966 us
Bilinear Interpolate 1: 33754 us
Blur layer 1, A: 220381 us
Blur layer 1, B: 224386 us
Matrx Subtract 1: 2375 us
Bilinear Interpolate 2: 87132 us
Blur layer 2, A: 428130 us
Blur layer 2, B: 417904 us
Matrx Subtract 2: 5238 us
Bilinear Interpolate 3: 160028 us
Blur layer 3, A: 959453 us
Blur layer 3, B: 953917 us
Matrx Subtract 3: 11744 us
Bilinear Interpolate 4: 362109 us
Find extrema: 74638 us
Gradient maps: 182613 us
Characterize keypoints: 7543 us
Draw keypoints: 43821 us
```

Fig 4. Timing output for the sequential implementation of the SIFT algorithm.

Test Cases

I found it was difficult to compare my results to others to check for correct results. This is due to the variability in the implementations and parameters of the algorithms, which are not generally shared. Instead, I will prepare images in GIMP by editing pixel values to make predictable gradients and difference-of-gaussian maps in order to verify the keypoint locations and orientations.

The other portion of the testing will look at the performance as the size of the input data and image pyramid layers is increased. The complexity of the algorithm with respect to the input image sizes is almost O(N), but the complexity with respect to image pyramid layers is almost $O(1.5^N)$. This comes from the exponentially larger images as the pyramid grows downward.

In total, I will test small contrived grayscale images made in GIMP to test functionality, 512x512 images from the ImageNET database with varying numbers of pyramid layers to test performance, and 1920x1080 images from my desktop webcam to test a realistic dataset.

References

[1] D. G. Lowe, "Object recognition from local scale-invariant features," Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999, pp. 1150-1157 vol.2, doi: 10.1109/ICCV.1999.790410.

[2] Y. Sa, "Improved Bilinear Interpolation Method for Image Fast Processing," 2014 7th International Conference on Intelligent Computation Technology and Automation, 2014, pp. 308-311, doi: 10.1109/ICICTA.2014.82.

[4] Sinha, Utkarsh. “Finding Keypoints.” *AI Shack*, https://aishack.in/tutorials/sift-scale-invariant-feature-transform-keypoints/.