Parallel SIFT

Aniruddha Acharya K
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, India
aniruddha753@gmail.com

***Abstract*—A number of computer vision and image processing algorithms rely on image features, and compute them in their preprocessing phase. Among the well known image features available in the literature, Scale Invariant Feature Transform (SIFT) is known to be highly accurate, and is widely used in certain applications. However, for real time applications and applications like large scale image search, SIFT is seldom used because of the huge time it takes to compute them. As an attempt to solve this, I have implemented a parallel version of SIFT using GPU. The results obtained show a speedup of 62.08318 for a 800 X 640 image. Moreover it is seen that the algorithm is highly scaleable with respect to the image dimensions.**

I. Introduction

Computer vision is getting increasingly popular through the last few years. Some of the applications of image processing is tracking moving objects, recognizing human faces, identifying anomalies in safety critical places like banks, and searching for images similar to a given image from a large database, to name a few. Most of the algorithms for these involve an image feature extraction step. Image features are interesting points or regions of an image, which are quite invariant to translation, rotation, or scale changes in the image. Of the many feature extraction algorithms, SIFT [1] is one of the highly accurate algorithms that produce point features.

An overview of the computational steps in SIFT is given below:

A. Scale space creation

The input image is repeatedly convolved with 2-dimensional gaussians with increasing variances. The original image with each of the convoloved images form levels of the 1st octave. Next, the original image is subsampled and the convolutions are done as in 1st octave. This is the 2nd octave. Similarly there are k octaves in the scale space.

B. DIfference of Gaussian

In this step, each of the neighboring pairs of scale space images are considered and their difference is taken. Difference of gaussian approximates laplacian of guassian function, which is known to give high output values whenever the image scale matches with the scale of the function. This is a useful tool for getting features that are scale invariant.

C. Extrema detection

For all pixels in all of the images of scale space, 27 neighbors in the top scale, 27 neighbors in the bottom scale, and 26 neighboring pixels in the current scale is checked for whether the point under consideration is a minimum or maximum. If it is an extremum, it is further checked if the point has a high contrast by checking whether the pixel value is greater than some threshold (0.3 used, suggested by original paper [1]) Further, it is verified whether or not the point is on an edge. If it is on an edge, it is not a stable feature and is ignored. If a point clears all these tests, that pixel is a SIFT keypoint

D. Orientation and descriptor creation

SIFT features have 5 characteristics: x, y, scale, orientation, descriptor vector. The details of this step is given in the methodology section.

Unfortunately sequential codes for SIFT are too slow for real time applications and large scale applications. I have used GPU to speedup the execution of SIFT.

II. Methodology

All the steps of SIFT were implemented on GPU.

Scale space construction involves implementing two functionalities:

A. Subsampling a given image

This is quite easily done on GPU by using statements like `out[threaIdx.x][threadIdx.y] = in[threadIdx.x/K][threadIdx.y/K]`, where `K = 2, 4 or 8 and so on`, and by using the right number of blocks and threads per block.

B. 2D convolution with a guassian kernel

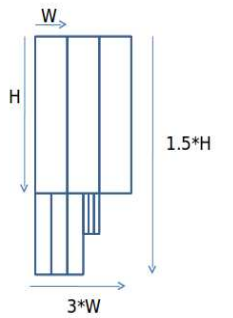
2D convolution is a time consuming operation. Fortunately gaussian kernel is a separable kernel, which means the 2D convolution can be replaced by two 1D gaussian convolutions, once for the rows and once for the columns. I have used Nvidia's separable convolution kernel for this step.

1) Optimization:

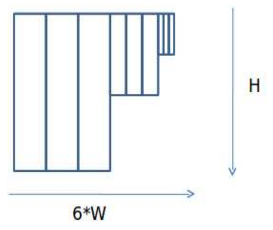
Convolution has to be done for all octaves and all levels. Since the number of blocks, number of threads, the kernel arguements are different for different octaves and levels, different kernel calls are used for each octave and level. This, also has been the approach used by [2].

However, it is possible to combine convolution of all octaves and all levels by using the following techniques:

- Decide the number of blocks of the combined kernel by placing individual threadblocks of different octaves and levels on the GPU grid.
  For instance, if convolution on rows requires `W X H` blocks for Octave 1, `W/2 X H/2` for Octave 2 and so on, fig1 and fig2 shows two ways of placing the threadblocks for all octaves and levels. Since fig 1 consumes lesser area on GPU, it was chosen for row convolution.



Area of grid used = 4.5 WH
Fig 1



Area of grid used = 6WH
Fig 2

- While calling the kernel, send parameters corresponding to all octaves and levels.

- In the kernel, define some parameter pointer variables so that this variable can be filled by the right parameters depending on the octave and level the thread belongs.

- Have a series of if-else conditions at the beginning of the kernel based on the placement of blocks to classify the thread into the right octave and level. Note that none of these if else conditions will lead to warp divergence because the conditions are only based on `blockIdx`.

  Ideally, since all thread blocks work parallely, by the time the first octave finishes the job, all the other octaves would have already completed their jobs, as the other octaves are of smaller size.
  This means we are completely hiding the time for octaves 2-6. But this does not completely happen as thread blocks on GPUs are not always parallel and they are time shared. Yet, there is a signigicant time advantage due to this, as shown in the results section.

This optimization of combining kernels is not restricted to convolution. It can be applied to any scenario where there are many problems of the same nature but with different input parameters, different sizes and different inputs. I have used this optimization for subsampling and for some later operations as well.

C. Difference of Gaussian(doG)

This can also be quite easily done by invoking a kernel which takes two of the adjacent scale space images and takes the difference. For performance, I applied the combined kernel optimization to this step too.

D. Subsampling some doGs

The step of extrema detection requires an additional doG image at each octave except Octave 1, that must got by subsampling the last level image of the previous octave. Combined kernel optimization was used here also.

E. Extrema detection

I used one kernel per image in the scale space. The kernel checks the neighbors in scale space for all pixels in the image. It also does the constrast and edge check as mentioned in the Introduction section.
If a point clears all these tests, it is updated in a global array. To avoid race among threads during array updation, atomicAdds are used. atomicAdd is a costly operation, but since practically all threadblocks do not execute in synchronism, and the number of keypoints is usually 1 order less than the image size, the number of threads simultaneously accessing atomicAdd will not be very high.
I did NOT use combined kernel optimization for this step because that creates much more contention for the global array due to atomicAdd.

F. Orientation and descriptor creation

Each keypoint was assigned a threadblock, having 16X16 threads. The threads load the image patch surrounding the keypoint.
Initially all threads find out the derivates at their corresponding location and convert them to polar co-ordinate (M, θ). $'\theta'$ is quantized into 36 bins from $-\pi$ to π. For getting orientation, we need to create a histogram of orientations. If we employ the method of each thread updating the appropriate bin independently, we will have synchronization problem. Hence the method of multiple histogram arrays are used.
Here, 32 (size of warp) histogram arrays are allocated. One warp is activated at a time and each thread is allowed to update into its own histogram. Then all the

32 histograms are added together and added to a final histogram.

Next maximum of the histogram is found by modelling it as REDUCE and by using the standard tree reduce. Similarly the second maximum is also found out(by setting the max to 0). In case the magnitude of the second maximum is greater than 80% of the magnitude of the first, an additional keypoint is created at the same location and scale, but with different orientation.

For the descriptor creation, the 16 X 16 patch is divided into 16 4X4 sub patches and a separate histogram is created for each of the subpatches, with 8 bins. Then, all the histograms are concatenated to form the histogram.

1) Optimization: I have used pinned memory for transfering image from host to device. Pinned memory is a kind of host memory allocated using cudaHostAlloc (instead of malloc), which is guarenteed that it will not be paged out or sent to second memory. cudaMemcpy works fast with pinned memory than normal memory. This is because for a normal memory, it is first transferred to a temporary buffer and then to device, but for pinned memory, it is directly send to device. But pinned memory is a limited resource and I could allocate a maximum of 640X640 floats.

The input image is split into 8-pixel overlaping tiles of 640X640. Each of the tiles is processed separately, in each iteration transfering the tile using pinned memory.

2) Optimization: This requires that before the start of next iteration, the pinned memory should have the fresh tile contents. Such a copy takes around 3360 microseconds. While one iteration of GPU takes 4046 microseconds. Since they both are almost the same, I chose to make the two tasks parallel by using pthreads.

III. Experiments and Results

Performance of optimizations

A. Combined kernel optimization (for image size of 4480 X 3200):

Total times for subsampling, row convolution, column convolution:

Separate kernels time = 59,863 microseconds
Combined kernel time = 33,482 microseconds
Gain of 26,381 microseconds
Around 14.49% of the total time

B. Pinned memory optimization (for image size of 4480 X 3200):

Without pinned memory: 81041.000000 microseconds
With pinned memory: 35101.000000 microseconds
Advantage: 45940 microseconds
Around 25.249 % of the total time

C. Pinned memory optimization (for image size of 2650 X 1920):

Without pinned memory: 30473.000000 microseconds
With pinned memory: 15064.000000 microseconds
Advantage: 15409 microseconds
Around 20.86 % of total time

I tested the implementation on 3 images of sizes 800 X 640, 2650 X 1920, 4480 X 3200. The run-time for serial code was obtained by using the siftpp code [3]. The speedup is shown in the graph.

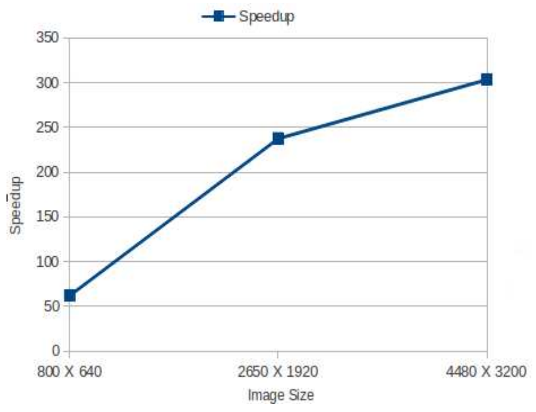


Image Area	Sequential SIFT run time	Parallel SIFT run time
800 X 640	1,474,910	23,757
2650 X 1920	17,542,699	73,843
4480 X 3200	55,200,716	1,81,943

TABLE I
Execution times in microseconds

IV. Conclusions

SIFT is an algorithm with high potential of parallelism. Since it extracts local features, there are no far off dependencies in the image space. But it is a very laborious task to port all of the algorithm on GPU. Some of the authors have ported only the convolution part and acheived a speed up of only 1.9X. On the contrary, my implementation used GPU for all of the algorithm. It produced a surprising speedup of 62.08. The novel optimization of combined kernel, the use of CPU using pthreads and pinned memory optimization have significantly contributed to the speedup.

Future work of this would be to extend this for videos.

References

[1] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," in *International Journal of Computer Vision, vol. 60, pp. 91110*, 2004.

[2] S. Warn, W. Emeneker, J. Cothren, and A. Apon, "Accelerating SIFT on Parallel Architectures," in *IEEE International Conference CLUSTER '09*, 2009.

[3] A. Vedaldi, "Sift++ source code and documentation [online]," in *http://www.vlfeat.org/vedaldi/code/siftpp.html*, 2009.