Make Scale Invariant Feature Transform “Fly” with CUDA

Yuhong Mo[1], Chaoyi Tan[2], Chenghao Wang[3], Hao Qin[4] and Yushan Dong[5]
[1]Carnegie Mellon University, Electrical and Computer Engineering, PA, USA
[2]Northeastern University, Electrical and Computer Engineering, MA, USA
[3]Georgia Institute of Technology, Computer Science, GA, USA
[4]Independent, CHINA
[5]University of Maryland, Machine Learning, USA

[1]Corresponding Author: yuhongmo@cmu.edu

Received: 01-05-2024 Revised: 16-05-2024 Accepted: 05-06-2024

ABSTRACT

This paper introduces an implementation of scale invariant feature transform (SIFT) algorithm with CUDA. Primary steps including building the Gaussian pyramid and the difference of Gaussian pyramid, identification, localization [1], and orientation generation of key-points are realized on GPU with CUDA. A detailed description of important kernel function implementations is covered along with optimizations made to achieve high performance, and a comparison between the CUDA version SIFT algorithm and a baseline sequential CPU implementation is included.

***Keywords*--** SIFT, CUDA, Parallelism

I. INTRODUCTION

In computer vision, a common task is to detect local features from images to match across images. On the one hand, we want the algorithm to perform well under image changes such as scaling, rotation, and different lighting. On the other hand, due to the increase in the resolution of images and sometimes the demand for real-time feature extraction and matching, it is desirable to have a sufficiently fast implementation.[2]

Therefore, in this project, we parallelize a classical feature extraction algorithm called scale invariant feature transform (SIFT). This algorithm has relatively good effects under affine transforms of images, noise, and lighting changes. With CUDA, we exploit GPU computation to accelerate a current CPU-only implementation.

II. PROCEDURES OVERVIEW

SIFT is a multi-step algorithm[3]. The primary steps for realizing a SIFT algorithm are listed as follows.

- **Building Gaussian Pyramid.** With the input image, we first build a pyramid of images by down-sampling by two at each time. For each group of images inside every pyramid level, its size is half of its previous group. The number of groups in total is O = log2 (min(M, N)) − 3 where M and N are the length and height of the original image, respectively. Inside each level of the pyramid, we create S copies of the same size. We then convolute a Gaussian kernel with each image in the pyramid, respectively.
- **Building Difference of Gaussian Pyramid[4].** After build- ing the Gaussian pyramid, we calculate the difference between each two neighboring images at the same level to get the difference of Gaussian.
- **Orientation Generation.** We count the directions of pixels in the neighboring circle of key-points to determine the orientation.[6]

In theory, there is one last step to generate key-point descriptors, but in our discussion, we leave out this step in both implementations. In the next sections, we discuss about detailed implementation of the GPU version[7].

III. GENERATION OF GAUSSIAN AND DIFFERENCE OF GAUSSIAN PYRAMID

The first part generates two pyramids of images that facilitate key-point identification.[8] The input image as a float matrix is first transferred to global memory. It is stored in row-major order. [9] It is then doubled in height and width with linear interpolation to be the bottom layer image of the first level of the Gaussian pyramid.[10] Then the Gaussian pyramid is built with the following rule: if it is the first layer in a level,[11] it is produced from the last layer of the previous level with width and height halved;[12] if it is not the first layer in a level, it is produced from its previous layer with a Gaussian blur applied[13]. A difference of Gaussian pyramid is then built by taking the difference of neighboring layers inside a level.

We note that size-doubling, size-halving, taking the difference, and Gaussian blurs are performed on GPU without data transferring from or to the CPU[14]. Every produced image is stored in global memory as a float array, [15] but between images there is no connection in memory location. In the following part, we discuss the implementation of the Gaussian blur kernel, which is the key function of this part.

The algorithm design finds the convolution output for this 32x32 brick using one thread block.

Diving deeper into the convolution operation for 2×S+1 filter size, to find the convolution output for the 32x32 bricks, we need additional S rows/columns on each side of the 32x32 brick so that we can find the convolution output for the pixels at the edges. For our convenience and further discussion, this 32x32 brick along with S additional rows and columns on each side is called a Tile. [16] This data in the Tile is used multiple times to find the convolution output for itself and its neighbors. Thus, it makes sense to copy the whole tile into the shared memory and make a Thread Block work on it since the shared data will be in the memory for all threads in a Thread Block to use. Thus, our one Thread Block handles the convolution output for one Tile.[17]

The first thing that the threads do is to copy the whole of the Tile into the shared memory, which then can be used for further computation parallelly by all threads. The exact details of how this is done are mentioned in Point #3 below. The threads in a block need to be synchronized for this purpose, and hence we use __syncthreads() at this point. This ensures that the convolution operation is only performed when all the data is copied into the shared memory and no garbage values are used in the convolution operation. Once all the data of a Tile resides

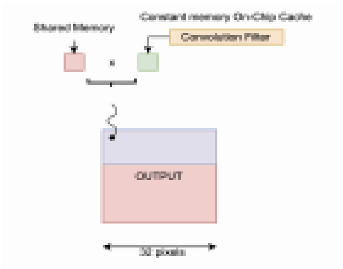


Figure 1: Performing Convolution Operation using threads

in the shared memory, we can proceed with performing the convolution operation. One thing to notice is that the Convolution Filter is something that is shared by the whole input matrix no matter which Tile we are finding the convolution for. Hence, it makes sense to cache the Convolution Filter in the Constant Memory On-Chip Cache[18]. Now for finding the output, each thread calculates the Convolution output for one output pixel, which means it iterates over the Filter sized sub-matrix in a Tile with the output pixel being at the center, multiplies the Tile values with its corresponding filter values, and then sums them up for finding the result for that output pixel. As already mentioned, each thread finds the output value for one convolution output. This is shown in Figure 1.

A key feature of the Gaussian Blur, which we feel contrasts our work to the work of peer teams is that we did not design our algorithm for only a specific filter size. This was also a requirement of the SIFT algorithm which we implemented. Thus, we had to decide what would be the maximum size of the filter that we would support. The value that we went ahead with was 71. The reasoning behind this value is explained in Point #1 below.

1) Determining the Maximum Supported Filter Size

Let us assume the brick size to be 32x32 for now (the rationale of it is presented in the below point number #2). The maximum shared memory available to us is 48k bytes. As already explained, the whole of the Tile needs to reside in the memory. Thus,

$$4 \times (32 + 2 \times S) \times (32 + 2 \times S) < 48k \qquad (1)$$

where S comes from the size of the filter which is $(2 \times S + 1)$ and a multiplicative factor comes from the size of a Float variable which is the type of the input to the Gaussian Filter. This approximately gives us a value of S to be: $S < 39$ and restricts the Maximum Filter Size to be less than 79. Taking a more conservative approach, we empirically determined the value of the Maximum Filter Size that we would support would be 71 instead of 77, as the higher values were running into memory issues even though theoretically they should not.

2) Determining the Brick Size

The rationale for choosing the brick size to the specific value of 32x32 instead of any other value is based on the Maximum Filter Size that we wanted to support. Had we chosen a brick size of 64x64 or higher, the size of the tile that would now have to be stored in the shared memory would have been:

$$4 \times (64 + 2 \times S) \times (64 + 2 \times S) < 48k \qquad (2)$$

which gives the following constraint on the value of S: $S < 23$ and restricts the Maximum Filter Size to be less than 47. This was something that we did not want to do and we wanted to support higher filter sizes for our Gaussian blur function. Hence, we chose the brick size not greater than 32 × 32.

Furthermore, keeping a smaller brick size would have not allowed us to use more number of threads, and our parallel computation by using multiple threads could have been restricted by the brick size itself which would not have allowed us to use the hardware resources to the best.

3) Copying the Data into Shared Memory

There are two ways of copying the data into the shared memory from the device memory since the value of the Filter Size $(2 \times S + 1)$ varies which makes S a variable. Now depending on the value of S, either we can keep the total number of threads fixed and then vary the number of rows that can be copied to the shared memory, or, we could vary the number of threads while keeping the number of rows fixed. We implemented and experimented with both these approaches and finally chose to make this part of the code configurable in case anyone is interested in trying out both approaches themselves. However, based on our results, there was nothing conclusive that could be reached about one method being better than the other in terms of time taken. Additionally, keeping the threads to a constant number introduced another hyperparameter of the number of threads that we could use, and again there were no conclusive results regarding that either. Keeping the number of threads constant seemed to work well with a lesser number of threads in the case of smaller images with smaller filters, whereas with an increase in the image/filter size (beyond a certain point), having more threads paid off, and that too the peak performance was observed with a different number of threads for different combinations of image sizes and kernel sizes. Additionally, the method of varying the number of threads by keeping the number of rows copied consistent seemed to work better than the above approach for larger images and/or larger filter sizes. Thus, we chose to go ahead with this approach.[19]

In spite of having both configurations possible as a part of our code base, we chose to perform our experiments on keeping the number of rows fixed and varying the number of threads on the basis of the size of the Gaussian filter used to blur the images. The numerical aspects of it are explained in the point right below (#4).

4) The Number of Threads

As explained in the above point (#3), not having concrete evidence of one number of threads working well for different combinations of images and kernel sizes, we chose to go ahead with using a number of threads that is dependent on the Gaussian Filter size. To fix the number of threads we want, we refer to the action of copying the image to shared memory. For this purpose, let h rows be copied into shared memory at once and using the fact that the maximum number of threads in a block is 1024:

$$(32 + 2 \times S) \times h < 1024 \quad (3)$$

But the maximum value of S comes from the Maximum Filter Size = 71 supported by our kernel, which gives us h ~ 10 and we chose:

$$h = 8 \quad (4)$$

Thus, the number of threads per thread block = $(32 + 2 \times S) \times 8$ where S comes from the filter size of $(2 \times S + 1)$.

5) The Number of Thread Blocks

Since the image is divided into multiple bricks of 32 × 32 size, each being handled by a Thread Block, the number of Thread Blocks is the same as the number of such bricks, which is:

$$(\text{Image\_\_Width} \times \text{Image\_\_Height})/(32 \times 32) \quad (5)$$

B. Implementation of Other Kernels

We briefly talk about other unimportant kernels including size-doubling, size-halving, and taking difference in this part.

- Size Doubling. For an input image with width and height, the output image has a size of twice of width and twice of height. Within each block, each thread is mapped to a column of input image. Each block reads lines of the input image coalescingly and writes the values into shared memory. After reading data, a patch of input image is present in shared memory. One thread then produces 4 outputs which are linear interpolations of its neighboring pixels, whose values are in shared memory. Since their access to shared memory has the same pattern, no bank conflict within the same warp will happen. Finally, the outputs are written to the output array. The thread number per block is set to be the maximum of 1024.
- Size Halving. This kernel decreases the input image size by half. Each thread in the block is mapped to one column of output value. Since there is no data sharing, a thread simply reads the desired value in the input image and writes to the output image. Reading inputs has a stride of 2, and writing to the output is coalescing. The thread number within each block is also set to be the maximum value of 1024 to fully utilize a block.[20]
- Taking Difference. This kernel takes two input arrays, calculates the difference between corresponding elements, and writes to the output. Each thread is mapped to one column of input and output data. Since there is no data sharing, each thread simply reads in two input values, taking their difference, and writes to the output for each position in a column. Both reading from the input and writing to the output is coalecsingly done. The thread number is set to be

the maximum value of 1024, since each thread uses less than 64 registers.

IV. EXTRACTION OF LOCAL EXTREMA

The task of this part is to identify and extract information about extrema from the difference of Gaussian pyramid.

A. Input

The input of this function are multiple levels of images. Specifically, within each level, the size of every image is the same. Between each level, from a front level to its next level, both the size and width are halved. The size of images represent the level of detail of images. Throughout sections 4 and 5, we use a specific setting with the bottom level having a size of 3840 × 2160, where the size is indicated as width×height. Note that its size is double the original input image. Within each level of difference of Gaussian pyramid, a total of 6 images are produced. There are 8 levels in total and images within each level have sizes of 3840 × 2160, 1920 × 1080, 960 × 540, 480 × 270, 240 × 135, 120 × 67, 60 × 33, and 30 × 16.

B. Output

The output destination of this function is a pre-allocated structure (init__feat) in global memory. Inside this structure, a count variable (initially 0) records the number of extracted extrema. Several arrays store necessary information that is required for identifying the extrema point and future processing, including level number, image index within the level, coordinates within the image, and values of the extremum's neighbors. The last term is required by later processing. By providing neighboring information, we avoid accessing global memory for values inside the pyramid in an irregular pattern. By storing information in a structure containing multiple arrays rather than an array containing structures of many data members, we make the access to the output by the next stage kernel coalescing.

C. Naive Implementation

To qualify as a local extrema, the pixel's position is not at the boundary. Here, boundary points include the edge positions within an image, and also the first and last image within a level. A local extrema's absolute value is greater than a threshold. If the value is less than 0, it is also less than the 8 pixels surrounding it within the same image as well as the 18 pixels that are from the image before it and the image behind it. If the value is larger than 0, it is then larger than all its neighbors. Therefore, the local extrema is calculated in a 3-D manner.[20]

In a naive manner, we let every thread examine a single pixel. Specifically, each thread block is of size 32×32 (since we want to maximize the number of threads). Within each kernel call, the thread reads in 3 pixel values: one from the previous image, one from the current image, and one from the next image. Since access to its neighbors is needed, three pieces of 32×32 float shared memory is used to store the value obtained. Then each thread examines whether the current pixel qualifies an extrema. If so, it calls an atomic add to count to preserve a location in the output structure. It then writes the necessary information and the kernel call is finished. For every level, one kernel call is used. For every patch in an image of a level, a block is launched. With previously specified settings, this implementation runs for 0.0043s on average.

D. Optimization

In order to improve the performance, several techniques are tried and tested. Here we discuss a few of them.

1) Improving Memory Reuse: One major drawback of the previous implementation is that the shared memory reuse is low. For each thread, the majority of work is spent on reading from global memory. Since we have many images inside the same level, an image can both act as a layer to be examined for extrema and the comparison image for the next layer. To be clearer, each thread can sequentially read in pixels from the first image inside a level to the last image of the level. Extrema are located and stored along the way. Therefore, for each image between the first and last image, the reading-in number reduces from 2 to 1.[20]

We also consider letting each thread process columns of pixels inside each image instead of one pixel. Threads within the same warp are still reading coalescing positions with each loading. As an unjustified choice of parameters, we let each thread examine thread height=16 pixels inside every image (but actually 18 pixels are loaded from global memory since the top and bottom pixels are also needed). We choose the block size to be thread number=128. In terms of shared memory, after careful arrangement, a minimum of 2 pieces of thread number × (thread height+2) float shared memory are needed, saving 1/3 compared with the naive implementation. Within each thread, local arrays preserve pixels' information that is exiled from shared memory (since other threads will not need it later) but is needed by the thread itself.[20]

In terms of resource utilization, register usage includes temporary numbers and 3 local arrays each of size roughly thread height floats. For a block, reg- ister numbers are roughly thread number × (3 ×thread height+other registers). The shared memory usage is 2×(thread height+2) ×thread number. With this setting, the runtime is 0.0035s on average.

2) Choosing Right Block Size: The previous change in code improves memory access but limits the number of blocks that are launched simultaneously. We improve work distribution in this part.

We first consider thread number. Ideally, one block can have at most 1024 threads. For bottom-level images with large sizes, a large thread number is reasonable, but for top-level images with smaller sizes, a large thread number leaves many threads doing no work. With that consideration, the thread number is chosen so that it is the minimum of image width and the default thread number. But for small images, shared memory is still wasted. Using an empirical approach, the performance is best when the default thread number is 256. In terms of the height of the column that each thread processes (we call it thread height), we want to maximize hardware resources. Within each SM, there are 1024 hardware threads. We want to use all these threads without exceeding a total of 64K registers. It turns out that when the thread height is 6, 64 registers are used by every thread, which is optimal. Empirical results proved this idea. With 256 threads in each block and every block processing 6 pixels (and loading in 8 pixels) in each image, the runtime is 0.0021s.

3) Using Streams: For each level of difference of Gaussian pyramid, one kernel call is launched. But these calls are not dependent on each other. With this observation, we create streams to perform these kernels concurrently. With streams, the runtime is 0.0019s. The usage of stream primarily improves the hardware utilization when the kernel is called on upper levels where the image size is small.

4) Reducing Atomic Operations: We rely on atomic operations to update the results. A single thread may produce multiple outputs, and thus keeping results locally and writing to the output structure in the end together can reduce atomic operations. As an experiment, we give each thread local variables to store one temporary result, so that the thread can keep the result locally and write to the output together with a second result. However, as the possibility of generating a local extremum is rather low for our case (<1%), a thread is unlikely to find multiple extrema in a single run. The runtime is worse due to extra registers that are required to locally store results, which limits the number of threads that can run concurrently in one SM.

5) Unrolling: Each thread iterates through columns whose height is pre-fixed. We specified pragma unroll to perform loop unrolling. But it is observed from register utilization that when thread height is set to be small (such as 6 in our case), the compiler has already performed this trick for us without explicit pragma.

E. Optimization Results

This part summarizes the performance of the local extrema extraction function. The result is based on the average runtime from 50 runs on the input specified in the previous section. In our final implementation, we adopt the code with the minimum runtime.

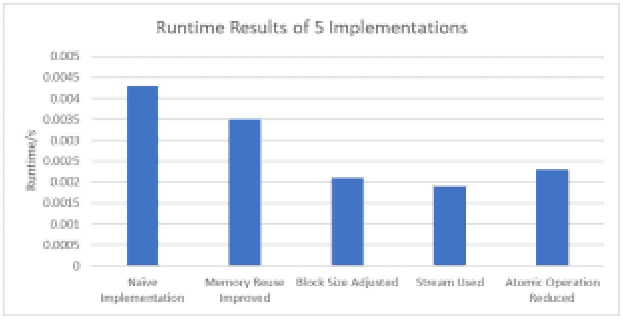


Figure 2: Runtime Results of 5 Implementations

V. KEY-POINT LOCALIZATION AND ORIENTATION GENERATION

We briefly describe these two parts for generating detailed information of key-points. We note that these two parts should be improved if time allowed.

A. Key-Point Localization

The input of this function is the output of extraction of local extrema. Each thread corresponds to one preliminary extremum. Using the memory layout described previously, each thread in the same warp accesses global memory coalescingly. Using each extremum's value and its neigh- bors' values, this kernel does a one-step approximation of the pixel's non-integer position. The core operation is solving a three-by-three linear system equation involving discrete approximation of the pixel's first and second order derivatives. The output is a structure similar to its input but with modified array members. Since some extrema will not qualify after this operation, a similar atomic add is used when a thread wants to output to global memory. Since this kernel function costs 26 registers, the thread number is set to be the maximum of 1024. With the same input as section 4, the average runtime of this function is 0.0015s.

B. Orientation Generation

Based on the output from key-point localization, this function identifies the orientations of key-points. It accesses a grid of pixels near it at the corresponding Gaussian pyramid. For each pixel in the neighborhood grid, its orientation is estimated by its local derivative. The number of pixels within each range of angle is accumulated. For our target pixel at the center, every angle range that has more pixels falling into, than its neighboring angle ranges counts as one valid angle descriptor. Therefore, for each input, many outputs can be generated.

The input of this function is the output structure of key- point localization. The output of the function is a structure containing the count and an array whose member is a structure similar to OpenCV's Keypoint data structure. The difference in data layout is because no kernels will further refer to the output from here so that memory coalesce is not needed. Each thread is mapped to one input key-point. To reduce atomic operations, local storage al- lows one key-point to be stored temporarily. Consequently, if two descriptors are generated, they will be written to output together by one atomic add. After reducing atomic add operations, the runtime of this function from the same setting as the previous part reduces from 12.3ms to 10.2ms.

The primary challenge of this part is that each thread's access to the Gaussian pyramid is irregular. The neighboring range for each pixel is also different depending on its level and layer, making it difficult to organize a nice memory access pattern.

VI. EXPERIMENTS

This section shows the overall performance of our GPU implementation against the baseline implementation.

A. *Baseline Code*

The baseline code is a CPU-only sequential implementation of SIFT algorithm with OpenCV. The complete version of baseline implements SIFT, generation of key- point descriptor, and key-point matching. For our purpose, we only care about the SIFT part.

B. *Architecture*

Both versions of SIFT implementations are tested on an ECE machine. Specifically, the platform where we run the code has a CPU Intel Xeon 4208, and a GPU NVidia Tesla T4. The Tesla T4 GPU is based on Turing architecture, with 40 streaming multiprocessors (SM) and 32 warps within each SM.

C. *Dataset and Experiment Setting*

The original dataset is an image of size 1920 × 1080 converted into a gray-scale float array. The CPU version starts with this array and outputs a vector of key-points. The GPU version first copies the float array into GPU global memory, generates an array of key-points, and copies key-points back to the main memory. But the first and last copies are not counted in the timing results.

To obtain input data of different sizes, we resize the original image with linear interpolation. The generation of input images is not timed in both implementations. We note that by default, the first operation that is done with the input image in both implementations is doubling its size, but in the result part, we still label the data size as the input image size. For each size, the program runs for 50 times to reduce error.

D. *Results*

For a more detailed comparison, the time for generating the Gaussian pyramid and the difference of Gaussian pyramid is recorded, and the time for key-point identification, localization, and orientation generation is recorded, separately.

We first show the runtime result for generating the Gaussian pyramid and the difference of Gaussian pyramid in the following figure. We note that in the GPU version, the generated pyramids are not copied back to the CPU.

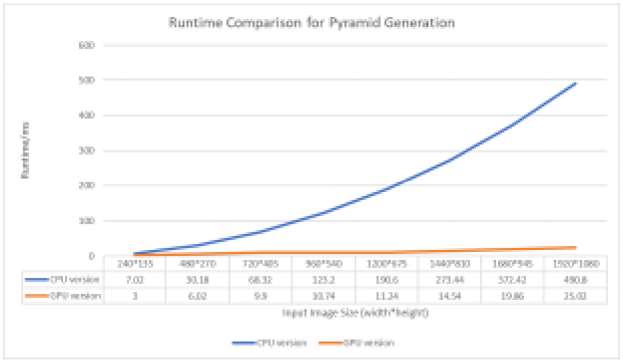


Figure 3: Runtime Comparison for Pyramid Generation

For the input size 1920 × 1080, the GPU version code is 20 times faster than the sequential CPU version.

The following figure shows the runtime result for key-point identification, localization, and orientation generation. For the input size 1920 × 1080, the GPU version

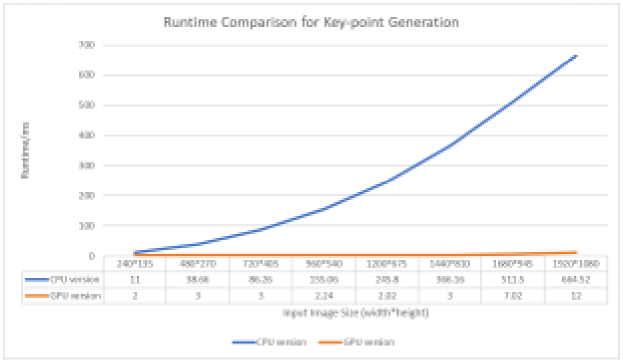
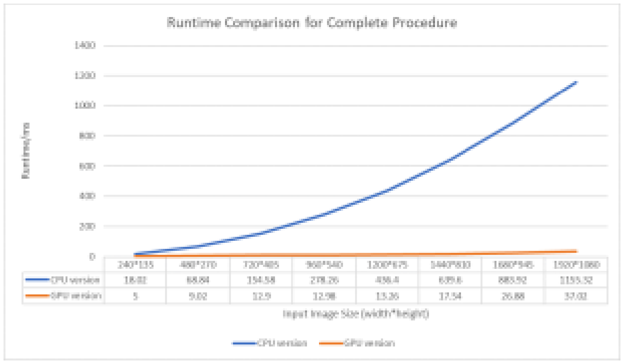


Figure 4: Runtime Comparison for Key-point Generation.

program is 50 times faster than the sequential version. We combine the two figures to get the runtime comparison for the complete SIFT procedure.



	240*135	480*270	720*405	960*540	1200*675	1440*810	1680*945	1920*1080
CPU version	18.02	68.84	154.58	278.26	436.4	639.6	883.92	1155.32
GPU version	5	9.02	12.9	12.98	13.26	17.34	26.88	37.02

Figure 5: Runtime Comparison for Complete Procedure.

Overall, for the input size 1920 ×1080, the GPU version is 30 times faster than the sequential version.

VII. CONCLUSION AND FUTURE WORK

In conclusion, we parallelize the SIFT algorithm with GPU computation using CUDA. For an input size of 1920 ×1080, our implementation runs 30 times faster than the CPU-only sequential version. In terms of Gaussian blur filter design, we have shown how our approach of dividing the input image into tiles and using shared memory helps to reduce memory access latency and improve the overall performance of the algorithm. We have also implemented several optimizations to improve the performance of our local extrema extraction function.

Yet there are still many things we can do. Specifically, we list a few below.

- In the generation of Gaussian pyramid, some kernels (taking the difference, zooming in/out) are doing very simple work. Ideally, we want to combine multiple calls of these simple kernels. On the one hand, it can reduce the number of kernel calls and therefore reduce the overhead. On the other hand, it can improve data reuse by avoiding constantly accessing global memory.
- The last stage of orientation generation exhibits an ugly pattern of global memory access, and the loop number inside each thread varies, making the work highly unbalanced between threads. Ideally, we should modify previous functions generating key-points so that in this kernel call, key-points within the same block are close to each other and similar in workload.
- For some usage of shared memory (such as in doubling the image), It is questionable whether using shared memory brings performance improvement, or a naive implementation may do a better job since it costs less resources. Although this kernel is not important to the overall performance, experiments regarding this question remain to be done.

REFERENCES

1. J. Jin, F. Ni, S. Dai, K. Li & B. Hong. (2024). Enhancing federated semi-supervised learning with out-of-distribution filtering amidst class mismatches. *Journal of Computer Technology and Applied Mathematics, 1*(1), 100–108.
2. S. Li, Y. Mo & Z. Li. (2022). Automated pneumonia detection in chest x-ray images using deep learning model. *Innovations in Applied Engineering and Technology*, pp. 1–6.
3. Z. Li, H. Yu, J. Xu, J. Liu & Y. Mo. (2023). Stock market analysis and prediction using lstm: A case study on technology stocks. *Innovations in Applied Engineering and Technology*, pp. 1–6, 2023.
4. K. Li, P. Xirui, J. Song, B. Hong & J. Wang. (2024). *The application of augmented reality (ar) in remote work and education*. arXiv preprint arXiv:2404.10579.
5. K. Li, A. Zhu, P. Zhao, J. Song & J. Liu. (2024). Utilizing deep learning to optimize software development processes. *Journal of Computer Technology and Applied Mathematics, 1*(1), 70-76.
6. T. Liu, S. Li, Y. Dong, Y. Mo & S. He. (2024). Spam detection and classification based on distilbert deep learning algorithm. *Applied Science and Engineering Journal for Advanced Research, 3*(3), 6–10.
7. Y. Mo, H. Qin, Y. Dong, Z. Zhu & Z. Li. (2024). Large language model (llm) ai text generation detection based on transformer deep learning algorithm. *International Journal of Engineering and Management Research, 14*(2), 154–159.
8. Y. Mo, S. Li, Y. Dong, Z. Zhu & Z. Li. 92024). Password complexity prediction based on roberta algorithm. *Applied Science and Engineering Journal for Advanced Research, 3*(3), 1–5.
9. J. Zhang, A. Xiang, Y. Cheng, Q. Yang & L. Wang. (2024). *Research on detection of floating objects in river and lake based on ai intelligent image recognition*. arXiv preprint arXiv:2404.06883.
10. J. Song, H. Liu, K. Li, J. Tian & Y. Mo. (2024). A comprehensive evaluation and comparison of enhanced learning methods. *Academic Journal of Science and Technology, 10*(3), 167–171.

11. A. Zhu, K. Li, T. Wu, P. Zhao & B. Hong. (2024). Cross-task multi-branch vision transformer for facial expression and mask wearing classification. *Journal of Computer Technology and Applied Mathematics*, *1*(1), 46–53.

12. Li, Huan, Feng Xu & Zheng Lin. (2023). ET-DM: Text to image via diffusion model with efficient Transformer. *Displays*, 80.

13. Lin, Zheng & Feng Xu. (2023). Simulation of robot automatic control model based on artificial intelligence algorithm. *2nd International Conference on Artificial Intelligence and Autonomous Robot Systems (AIARS)*.

14. Qiu, Shushan, et al. (2022). Day-ahead optimal scheduling of power–gas–heating integrated energy system considering energy routing. *Energy Reports*, *8*(2022), 1113-1122.

15. Chen, Jinfan, et al. (2023). Stochastic planning of integrated energy system based on correlation scenario generation method via Copula function considering multiple uncertainties in renewable energy sources and demands. *IET Renewable Power Generation 17*(12), 2978-2996.

16. Chen, Jinfan, et al. (2024). Reinforcement learning based two-timescale energy management for energy hub. *IET Renewable Power Generation 18*(3), 476-488.

17. Zhan, Rongrong, et al. (2023). Operation strategy of energy router considering compressed air energy storage. *4th International Conference on Advanced Electrical and Energy Systems (AEES)*.

18. Chen, Jinfan, et al. (2023). Robust optimization based multi-level coordinated scheduling strategy for energy hub in spot market. *7th International Conference on Green Energy and Applications (ICGEA)*.

19. Li, Zhuoying, et al. (2024). *AD-aligning: Emulating human-like generalization for cognitive domain adaptation in deep learning*. arXiv preprint arXiv:2405.09582.

20. Meng, Yinan, et al. (2023). Spring-IMU fusion-based proprioception for feedback control of soft manipulators. *IEEE/ASME Transactions on Mechatronics*.