EgpuIP: An Embedded GPU Accelerated Library for Image Processing

Luhan Wang[*†], Haipeng Jia[*‡], Yunquan Zhang[*], Kun Li[§], Cunyang Wei[*†]
[*]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[†]School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
[§]Microsoft Research
{wangluhan21s, jiahaipeng, zyq, weicunyang20g}@ict.ac.cn, kunli@microsoft.com

Abstract—**With the advances of embedded GPUs' programming models like GLES and OpenCL, the mobile processor has gained more parallel computing capability, which enables real-time image processing on portable devices. GLES is an excellent option to implement high-performance image processing on embedded GPUs due to its low hardware overhead. However, most GLES studies only focus on porting specific algorithms to embedded GPUs without a general optimization guide. In order to address the pending problems, this paper presents effective performance optimization chains to guide the optimization of image processing algorithms by using GLES. The image processing algorithms can be divided into three modes including data-independent, data-sharing and data-related according to their memory access and calculation characteristics. Based on this classification, our optimization chains contain four optimization directions. 1) Optimizing access to off-chip memory for the memory-bound data-independent algorithm; 2) Exploiting data locality by utilizing shared memory and cache for the data-sharing algorithm; 3) Redesigning the algorithm to optimize the sharing of computational results between threads for the data-related algorithm; 4) Making full use of computation resource for the above three algorithms. Based on these optimization chains, we design an embedded GPU accelerated image processing library, EgpuIP. To evaluate specific optimization methods and performance improvements, we employ histogram equalization, Gaussian pyramid and integral filter as the representative algorithm from EgpuIP and adequately optimize them guided by the proposed optimization chains. Compared to the OpenCV, experiments show that the three algorithms in EgpuIP provide up to 19×, 88×, and 3× speedup respectively.**

Index Terms—**Embedded GPU, Image Processing, Optimization Chain, Parallel Strategies**

I. Introduction

Recently, more and more scientists have adopted GPUs as computing accelerators due to their increasing computing power and programmability [1]. GPUs applied with mainstream programming models such as OpenCL [2] and CUDA [3] have provided significant speedup on a wide range of computer vision algorithms. However, this success has mostly been confined to desktop applications because most of these APIs are unavailable on mobile devices. With the widespread use of mobile devices, it is becoming increasingly important to implement real-time image processing on low-power devices, forcing us to use the Open Graphics Library for Embedded Systems (GLES) [4]. Nowadays, GLES allows general-purpose computing with compute shader [5]. Compute shader has a dedicated single-stage pipeline that executes independently of the rest of the graphics pipeline, which reduces the hardware overhead.

Most current mainstream open-source computer vision libraries do not support GLES due to its limitation of only being applied on portable devices. Because of the wide variety of image processing algorithms, GLES-related studies mainly address the porting of specific algorithms on embedded GPUs without providing generalized guidelines for optimizing extensive algorithms. Therefore, it is of great practical importance to propose an optimization strategy for image processing algorithms by analyzing and summarizing the essential features of image algorithms.

According to the memory access and calculation characteristics of image processing algorithms, we can divide them into data-independent, data-sharing, and data-related algorithms. For each of the three types, we propose off-chip memory access, data locality, communication and computation optimization chain separately. Based on these optimization chains, we design a real-time image processing algorithm library using comepute shader technique called EgpuIP for embedded GPUs. EgpuIP includes real-time implements of resizing, filtering, transposing, and sharpening, which are commonly used in deep learning image preprocessing pipelines. It is of practical significance to optimize these algorithms on embedded GPU.

To verify the effectiveness of the performance optimization chains, we employ histogram equalization, Gaussian pyramid, and integral filter as the representative algorithms from EgpuIP. We reduce the algorithm's off-chip memory bandwidth consumption through histogram localization for the data-independent histogram equalization algorithm. For the data-sharing Gaussian pyramid algorithm, we split the algorithm by row and column, respectively, and use shared memory to improve the reuse rate of on-chip hardware resources when filtering the image in the row direction. For the data-related integral filter algorithm, in addition to the methods mentioned above, this paper uses the three-stage parallel prefix-sum algorithm to optimize the sharing of computational results between threads and hence reduce the number of instructions. Experiments show that the three algorithms achieve $6.1\times$,

[‡]Corresponding authors.

$29.9\times$, and $2.2\times$ performance speedups on average compared to their respective OpenCV version when processing images range from 512×512 to 3840×2160.

Our key contributions are highlighted as follows:

- We propose practical performance optimization chains to address the lack of general optimization guidelines for porting image processing algorithms to embedded GPUs.
- Based on GLES, we present optimization methods for fully accelerating the three typical image processing algorithms of different patterns.
- We implement a high-performance embedded GPU library, EgpuIP, for many commonly used image processing algorithms based on these optimization chains.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III introduces the compute shader and describes the three algorithms in detail. Section IV presents the proposed performance optimization chains. Section V describes the specific optimization methods in the optimization chain when solving three typical algorithms. Section VI includes the experiments and discussions. Finally, the conclusion of this paper will be presented in the last section.

II. Related Work

With the emergence of general-purpose computing on embedded GPUs and their programming models like GLES and OpenCL, mobile processors are gaining a more parallel computing capability. However, the long processing time and high energy consumption of some image processing algorithms prevent them from being used effectively in real-time mobile applications [6].

Recently, GPUCV [7], MinGPU [8], and OpenVIDIA [9] have emerged as open source image processing and computer vision libraries based on the GPGPU technique. However, they are mainly targeted on the PC platform using interfaces such as CUDA, which are not available on the newest generation of handheld GPU [10]. The Open Source Computer Vision Library (OpenCV) uses the OpenCL programming model to accelerate image processing on embedded GPUs [11]. Compared with OpenCV, GLES does not yield a big apk file when implemented during the application development [12]. GLES is capable of supporting programmable shaders. Most recent research benefits from this release and takes advantage of the computational power of mobile battery-powered graphics processors. However, most of the research has focused on using parallel computing on GPUs with the GLSL tools to implement certain specific algorithms, such as corner detection [13]–[15]. The optimization methods proposed in these studies only work for specific algorithms and are not generalizable to handle other image processing algorithms.

In our work, we classify image processing algorithms according to their computational and memory access characteristics to propose a strategy for image processing optimization on embedded GPUs using GLES compute shader. Furthermore, the performance improvement is verified on the mobile platform.

III. Background

A. GLES Compute Shader

The compute shader technology is an approach to take advantage of the computational power of the GPU to implement GLES. Compute shaders are executed in workgroups. The compute shader is called once by each work item in each local workgroup in the global workgroup. Local workgroups support up to 128 work items on the hardware platform used in this paper. This way of working is illustrated in Fig. 1.

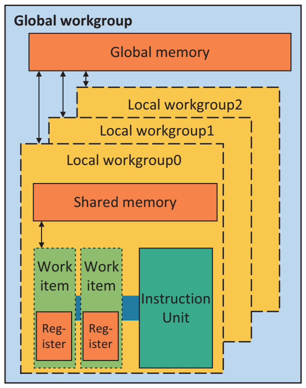

Fig. 1. Compute shader thread organization.

In this figure, each global workgroup contains three local workgroups, and each local workgroup contains two work items. Each work item has a unique three-dimensional global index (the second and third dimension are set to 0 at this point). Each shader's execution unit is essentially independent and can run in parallel on OpenGL-enabled GPU hardware. This differs slightly from CUDA and OpenCL, which support wavefront/warp. Therefore, there is no need to consider bank conflict when designing optimization strategies on the compute shader. The shared memory supported in the compute shader can be used to communicate between work items in the same local workgroup. Since the shared memory is typically backed by either cache or specialized fast local memory [16], the latency of accessing shared memory is much lower than accessing image textures or storage buffers. The compute shader also supports synchronization between work items, atomic operations on variables, etc. These techniques will be used frequently in subsequent algorithm optimizations.

B. The Histogram Equalization Algorithm

The image histogram represents the distribution of the image by quantifying the number of pixels per intensity value. For example, for a grayscale image, the histogram generation algorithm counts the number of times each pixel value (from 0 to 255) appears in that image and generates an array called a histogram containing 256 elements.

Histogram equalization is a method in image processing that improves the contrast in an image using the image's histogram. It is one of the most common algorithms to perform contrast enhancement [17]. Useful applications of Histogram Equalization enhancement include medical image processing, speech recognition and texture synthesis [18].

Equalization means changing the original histogram by spreading out the pixels that were previously concentrated on specific pixel values. A more rigorous representation is that for the original histogram $H(i)$, the equalized distribution $H'(i)$ is given by Eq. (1).

$$H'(i) = \sum_{0 \leq j < i} H(j) \tag{1}$$

Finally, we use $H'(i)$ to calculate the image, as shown in Eq. (2).

$$equalized(x, y) = H'(src(x, y)) \tag{2}$$

C. The Gaussian Pyramid Algorithm

1) Gaussian noise and Gaussian filtering: Noise is usually represented on an image as isolated pixels that cause strong visual effects but are not relevant to the object to be studied. Gaussian noise refers to a class of noise whose probability density function obeys a Gaussian distribution. Gaussian filtering is a linear smoothing filter applied to eliminate Gaussian noise and is widely used in the noise reduction process of image processing.

The conditional probability density function of the two-dimensional Gaussian distribution is shown in Eq. (3).

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

In practice, two-dimensional discrete Gaussian functions are commonly used as smoothing filters. A Gaussian filter is obtained by extending the binomial filter [19] to 5×5 and normalizing it. Eq. (4) and Eq. (5) are the matrix representations of the binomial and Gaussian filters respectively.

$$binomial = \frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{4}$$

$$gaussian = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{5}$$

For the Gaussian filter, the center value of the convolution kernel in Eq. (5) is the largest, and the surrounding values gradually decrease. Furthermore, the effect of Gaussian filtering is better than that of mean filtering.

2) Gaussian pyramid: A Gaussian pyramid is a collection of images. Specifically, with successive downsampling and Gaussian filtering, the images on the Gaussian pyramid gradually decrease in resolution as the layers increase until a specific termination condition is reached. The principle of the Gaussian pyramid is shown in Fig. 2.

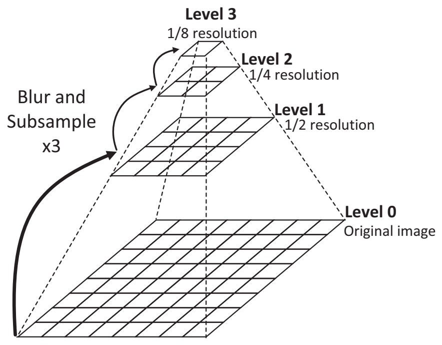

Fig. 2. Gaussian pyramid schematic.

D. The Integral Filter Algorithm

The integral filter algorithm filters an image to get its integral image and is widely used in applications with strong real-time requirements such as fast feature detection. An integral image $I(x, y)$ is the sum of original image pixels $i(x, y)$ at the left and top of the point (x, y) [20]. Its equation is shown in Eq. (6).

$$I(x, y) = \sum_{x'}^{x} \sum_{y'}^{y} i(x', y') \qquad (x' \leq x, y' \leq y) \tag{6}$$

Once we compute the integral image, we can find the sum of pixels in any rectangular region consisting of the upper left point (x_1, y_1), and the lower right point (x_2, y_2) with $O(1)$ time complexity.

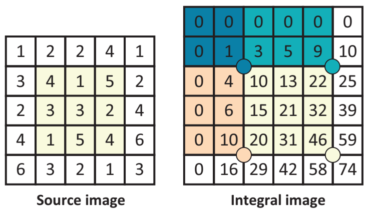

Fig. 3. Principle of integral image for fast feature calculation.

As shown in Fig. 3, eight summation operations $(4+1+5+3+3+2+1+5+4)$ are required to compute the rectangular region in the left source image, and the computation increases as the matrix region expand. The procedure is shown in Eq. (7).

$$S(x_1, y_1, x_2, y_2) = \sum_{y=y_1}^{y_2} \sum_{x=x_1}^{x_2} i(x, y) \tag{7}$$

In contrast, for the integral image on the right, only one addition and two subtractions $(46 + 1 - 9 - 10)$ are required to find the region sum, and the amount of computation does not increase with the matrix region. The process is shown in Eq. (8).

$$S(x_1, y_1, x_2, y_2) = I(x_1, y_1) + I(x_2, y_2) - I(x_1, y_2) - I(x_2, y_1) \tag{8}$$

In face detection [21], Haar-like features are extracted in this way.

IV. Performance Optimization Chains

According to the memory access and calculation characteristics, we divide image processing algorithms into three categories and propose effective performance optimization chains to provide practical programming guidance for embedded GPU image processing developers. The schematic diagram of the three categories is shown in Fig. 4. The three algorithms introduced above belong to the three categories, which cover most algorithms for image processing. Each optimization chain consists of some critical methods, which are organized in order according to the magnitude of their performance impact.

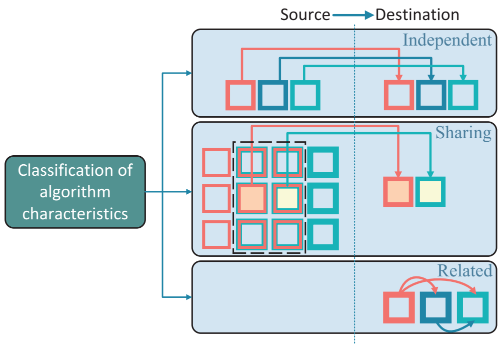

Fig. 4. Classification of algorithms in optimization.

A. Off-chip Memory Access Optimization Chain

In the first classification in Fig. 4, the pixel mapping at a certain coordinate is not affected by other pixels. We define this algorithm that satisfies the completely independent mapping of each pixel as a data-independent algorithm. Some of the simple image geometry transformation algorithms, image transposition, and color space transformation algorithms fall into this classification.

The off-chip memory access optimization chain represents a set of optimization methods used to optimize the utilization of off-chip memory bandwidth. We defined optimization spaces as follows: 1) Coalesce global memory access requests as much as possible through continuous alignment memory access and vector access. 2) Reduce the program's dependence on off-chip memory bandwidth by utilizing middle-tier caches with lower access latency. This chain is mainly used in the process of optimizing the data-independent algorithms.

B. Data Locality Optimization Chain

As shown in the second classification of Fig. 4, we define such algorithms that require sharing multiple source image pixels when computing a destination pixel as data-sharing algorithms. A large number of filtering, convolution, interpolation, and edge detection algorithms are in the scope of this type.

The data locality optimization chain is defined as a set of optimization methods that can efficiently access reusable data, which contain two optimization aspects: 1) Data reuse: Discover explicitly or implicitly reusable data during the operation of the algorithm. 2) Data locality: Placing data that is heavily reused in subsequent operations in shared memory. This chain is mostly applied to the optimization of data-sharing algorithms.

C. Communication Optimization Chain

In the data-related algorithm, the computation of one destination pixel uses the computation result of another destination pixel. The third category in Fig. 4 illustrates this relationship. We can define the communication optimization chain as a set of optimization methods that share computational results between threads. The optimization chain often requires redesigning the algorithm to parallelize the sharing of results between threads. For the most part, it is used in the optimization of data-related algorithm.

D. Computation Optimization Chain

Furthermore, we propose the computation optimization chain, which can achieve more extreme performance for the three types of algorithms mentioned above. The computation optimization chain is defined as a set of optimization methods that can make full use of computation resource, which contain three optimization aspects: 1) Improving thread-level parallelism. 2) Reducing dynamic instruction count per thread. 3) Instruction selection optimizations.

V. Optimization Case

A. Optimization of Data-Independent Algorithms

The histogram equalization algorithm can be divided into three main steps: histogram generation, array equalization, and image equalization.

1) Coalesce global memory access requests: The array equalization of the grayscale image contains only 256 elements. This process can be computed by enabling a single thread or on the CPU. For image equalization, the algorithm's pixels are independent and parallelizable, making it ideal for massively parallel GPU processing. The attention of image equalization optimization should be paid on fully exploiting memory bandwidth. Since the image textures of a single channel in the compute shader are not stored contiguously in memory, vectorizing the access memory to improve the off-chip memory bandwidth utilization is hard to achieve.

2) Reduce dependence on off-chip memory bandwidth:

a) Optimization analysis: The histogram generation operation count the number of pixels on each pixel value. It is theoretically well parallelized because each pixel value's counting are independent and can be assigned to different threads for parallel processing. However, there are serious concurrency conflicts among threads, which make us resolve the access conflicts by atomic operations, which achieve mutually exclusive protection of variables shared by multiple threads. The atomic operation of global memory makes the access memory serialized, which seriously affects the algorithm performance. We reduce the atomic operations on global memory and improve the concurrency between threads by histogram localization. In addition, this paper improves the utilization of GPU computing resources by selecting the optimal number of workgroups to achieve the best balance between counting and reduction.

b) Histogram localization: Accessing global memory is very costly, as local memory access may take only a few clock cycles, while global memory takes hundreds of clock cycles [22]. In the basic GPU implementation of the histogram generation algorithm, threads perform atomic operations on the histogram array stored in global memory. Due to the sequential access to the histogram array by threads and the high access latency of the global memory, many threads are left idle. We chunk the images so that each piece is processed by a workgroup and stores its local histogram in each shared memory. We refer to this operation as histogram localization. This method divides the histogram generation algorithm into two steps: counting and reduction. The process of counting and reduction is shown in Fig. 5.

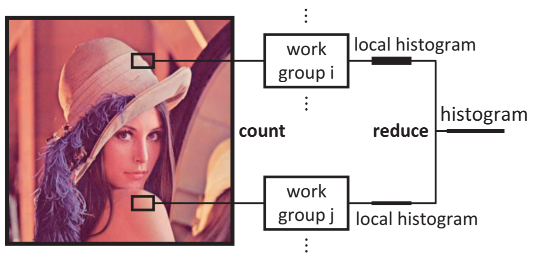

Fig. 5. Counting and reduction of histogram generation algorithm [23].

The data from each block is processed in parallel between workgroup during the counting step and written to global memory using atomic operations on a per-block basis in the final reduction step. This approach dramatically reduces atomic access to global memory and improves the algorithm's utilization of on-chip memory bandwidth.

c) Select the optimal number of workgroups: The number of workgroups needs to be well traded off for the two phases of counting and reduction. If the number of workgroups is large, the amount of data computed within each workgroup in the counting phase is small correspondingly, which leads to a large amount of data needing to be processed in the reduction phase. There are different optimal workgroup numbers for different image sizes. The optimal number of workgroups is adequately measured to be 256 for the image resolution of 1080P.

B. Optimization of Data-Sharing Algorithms

Gaussian pyramid generation is multiple executions of the pyramid down operation. Therefore, this algorithm's optimization mainly focuses on optimizing the pyramid down operation. We minimize the number of computational instructions by combining the downsampling and filtering and significantly reducing the access latency by separating the filtering in the row and column directions. The process is shown in Fig. 6.

1) Data reuse and data locality: We separate the 5×5 filter kernels into two one-dimensional filter kernels in the row and column directions, respectively. As shown in Fig. 6, the algorithm filters the 5×5 image matrix in the column direction to obtain a 1×5 matrix and then performs the row direction filtering to obtain the final value. In this process, the generated 1×5 matrix is heavily reused in subsequent calculations. For example, the 1×5 matrix obtained when calculating the second target pixel has three reused elements with the first 1×5 matrix. The memory access latency can be greatly reduced if the heavily reused elements are stored in the on-chip shared memory. In our work, the data after filtering by column is stored in the shared memory, and subsequent filter-by-row operations can access the data from the shared memory with much lower latency. This optimization strategy can significantly enhance the improved reuse of on-chip hardware resources.

2) Reduce the operation instructions:

a) Operation fusion: The algorithm uses a Gaussian kernel for image convolution and removes all even rows and columns to achieve downsampling. The number of computational instructions can be significantly reduced if the Gaussian kernel is convolved in interval rows and columns.

b) Reduce branch instructions: We reduce the number of dynamic instructions by eliminating conditional branches. In the Gaussian pyramid, conditional branching occurs when we compute the boundary elements. Two main ways are suitable for computing the boundary elements. One is to fill the boundary before uploading the image to the GPU. The other method uses a judgment statement in the kernel to handle the boundaries. The first approach simplifies the GPU kernel but increases the consumption of valuable memory resources on mobile devices. We adopt the second approach and use the ternary operator instead of conditional judgments. Since there is no access instruction in the ternary operator, the statement will be compiled into a single instruction. Therefore, in the boundary processing method of this paper, the address of the boundary data is first calculated using the ternary operator, and then the boundary data is loaded into the program in unison with the non-boundary data. Experiments show that the performance of boundary processing with ternary operators is improved by an average of 10%, which will be demonstrated in Section VI.

C. Optimization of Data-Related Algorithms

We decompose the integral filter algorithm by row direction and column direction. The row direction algorithm is the exclusive scan operation, as shown in Eq. (9).

$$[a_0, a_1, \cdots, a_{n-1}] \rightarrow [0, a_0, (a_0 + a_1), \cdots, \sum_{i=0}^{n-2} a_i] \qquad (9)$$

When implemented on GPU, we further decompose the column direction's integral filter algorithm into transpose and

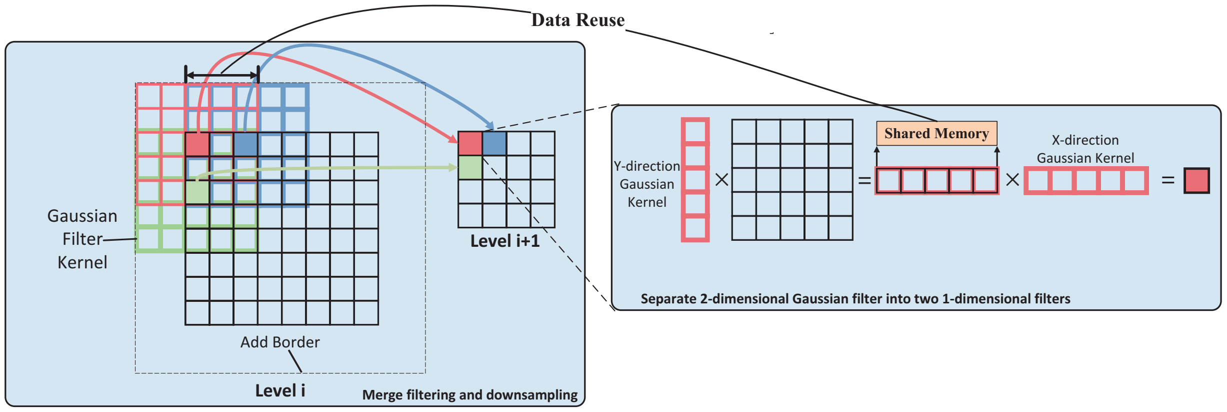
Data Reuse
Gaussian Filter Kernel
Level i+1
Add Border
Level i
Merge filtering and downsampling
Y-direction Gaussian Kernel
Shared Memory
X-direction Gaussian Kernel
Separate 2-dimensional Gaussian filter into two 1-dimensional filters

Fig. 6. Optimization of pyramid down.

exclusive scan operation to improve memory access performance. The integral filter algorithm is finally transformed into two exclusive scan operations and one transpose operation. The transpose algorithm is typically memory bound, so the optimization focuses on fully exploiting memory bandwidth. This optimization method has been introduced in the previous section and will not be repeated here. This section focuses on optimizations for other stages. For exclusive scan operations, we use a three-stage parallel prefix-sum algorithm based on a binary tree [24]. To fully utilize the on-chip memory bandwidth, the operations are performed in place using shared memory.

1) Redesign algorithm to parallelize the sharing results:

a) A three-stage parallel prefix-sum algorithm based on binary trees: The improved prefix-sum algorithm used in our work redesigns the naive prefix-sum algorithm. The algorithm performs a total of $2 * (n-1)$ addition operations and $(n-1)$ swap operations with a complexity of $O(n)$. It increases the total workload linearly with the growth of data elements, reduces each thread's computation instructions, and eventually enhances the program's performance. The specific flow of the algorithm is shown in Fig. 7.

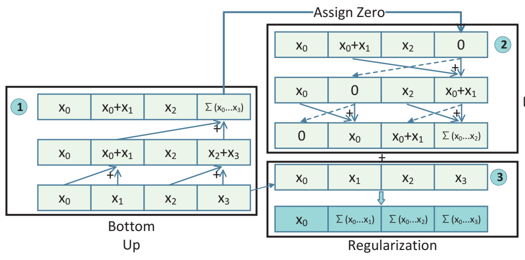

Fig. 7. The three stages of the parallel prefix-sum algorithm [25].

In Fig. 7, a bottom-up traverse is performed in box 1. After doing that, we assign 0 to the last node. Then we do a top-down traverse in box 2. After the top-down traverse, we add the array with the original array correspondingly, which shows in box 3. At this point, we have obtained the prefix sum of a 4-element array. Limited by the hardware platform, our kernel cannot scan arrays with sizes larger than 256. In our work, the number of threads per block is 128, and a single thread processes two elements.

b) Scan the whole image: For a specific n-element size row of the image, we launch local workgroups of n/b and $b/2$ work items per group. In our work, the size of b is 256.

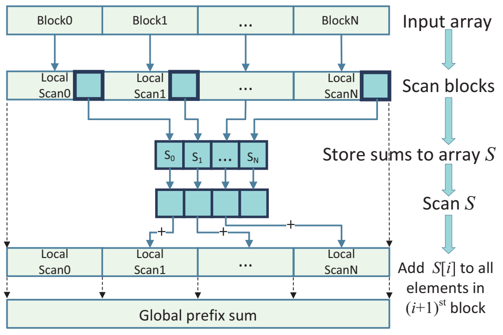

Fig. 8. Row direction prefix sum schematics [25].

As shown in Fig. 8, each workgroup scans a data block to get a local scan block. The last element of each local scan block is the local sum of the current data block, which we load into the array S. Then, we scan the array S and get the prefix sum of each local scan block. At this point, we add $S[i]$ to each element of the $(i+1)_{th}$ local scan block to get the prefix sum of the whole row.

Each row can be considered a separate array and scanned in parallel. In addition, we also parallelize each 256-sized block of data in each row. For an image, we launch a scan kernel using a grid with dimensions $n_{seg} \times h$, where n_{seg} is the number of segments in each row, and h is the height of the image. In order to solve the problem that the last block in each row is not a power of 2 in size, padding is performed before scanning.

2) Use higher throughput instructions: GPUs have different instructions with different throughput because of the distinctions in the number of functional units and the intrinsic characteristics of instruction self [26]. Selecting computation instructions with lower latency and higher throughput can effectively improve the performance of GPU programs. We use bitwise instructions instead of multiplication and division instructions for all power-of-2 operands, and the former has about five times higher throughput than the latter.

VI. Performance Evaluation

We select the CPU version function in OpenCV 4.5, the most famous computer vision library, to compare the performance and validate the correctness of our own GLES version.

A. Test Environment

We test the performance of EgpuIP and OpenCV on Mali-G77 GPU platform, the specific platform parameters are shown in TABLE 1.

TABLE I
EXPERIMENTAL PLATFORM SPECIFICATIONS

Specs	Value
GPU	Arm Mali-G77 MC9
Memory	12GB RAM + 256GB ROM
Architecture	Valhall
L2 Cache	512KB – 2MB
Bus Interface	AMBA 4 ACE, ACE-LITE
Scalability	7 to 16 Cores
OpenCV	4.5.5
OpenGL ES	3.2

The total storage size for all shared variables in a compute shader is defined by 32768 bytes. The product of the X, Y, and Z components of the local size must be less than 128, and the number of work groups that can be dispatched in a single dispatch call is defined by 65535 in X, Y, and Z dimensions.

B. Result and Analysis

The benchmark of an algorithm is its serial implementation on CPUs. The test interface in this paper is oriented to unsigned char, single channel images. The performance results of histogram equalization, Gaussian pyramid, and integral filter algorithms are shown in Fig. 9, Fig. 10, and Fig. 11, respectively. The bar chart reflects the time-consuming (in milliseconds) of the three implementations, and the two speed-up ratio line graphs respectively show the performance improvement of EgpuIP and OpenCV compared to the benchmark.

Our histogram equalization implement achieves an average of 6.07 times speedup compared with OpenCV and 18.74 times speedup compared with the benchmark. Compared with the OpenCV Gaussian pyramid algorithm, we get 29.93 times speedup on average. Moreover, the integral image algorithm in EgpuIP obtains 9.09 times speedup on average compared with the sequential algorithm running on the CPU and 2.21 times speedup compared with OpenCV, as shown in Figure 11.

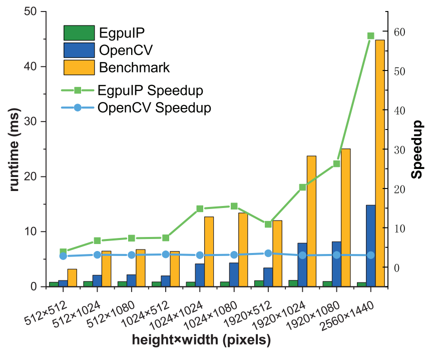

Fig. 9. The histogram equalization performance on Arm Mali-G77 GPU.

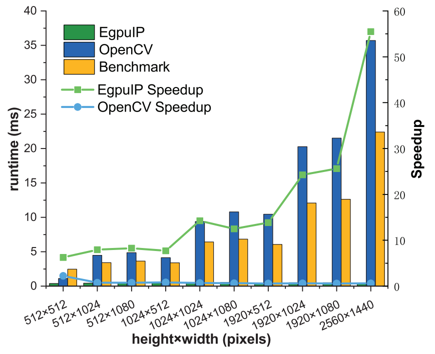

Fig. 10. The Gaussian pyramid performance on Arm Mali-G77 GPU.

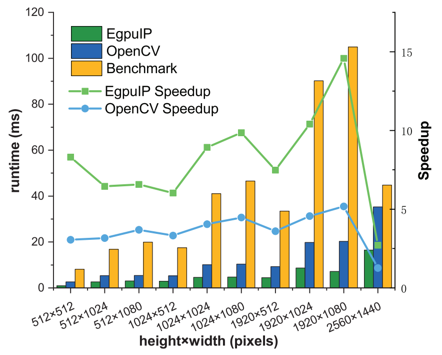

Fig. 11. The integral filter performance on Arm Mali-G77 GPU.

The specific performance improvements we discuss in the reducing branching instructions section in Section V are shown in the Fig. 12.

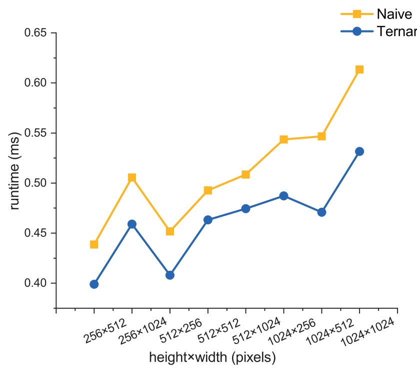

Fig. 12. Performance improvement after reducing the number of boundary processing instructions.

VII. Conclusion

In this paper, we propose practical performance optimization chains for image processing on embedded GPUs and present the optimization of three classical algorithms based on these chains. The effectiveness of our proposed optimization chains is verified by performance improvement analysis.

Most of the data-independent algorithms are memory-bound, the key to optimization is reducing the off-chip memory bandwidth consumption. Data-sharing algorithms are designed to reuse many source image pixels, so the key to optimization is data locality to minimize the access latency caused by data reuse. By splitting algorithms, we can further improve data reusability. Data-related algorithms are characterized by the fact that the computation of one destination pixel uses the computation result of another destination pixel. This optimization often requires redesigning the algorithm to share results across threads and achieve good parallelization among threads.

Acknowledgment

We would like to express our gratitude to all reviewer's insightful and valuable comments and suggestions for helping us polish this article. This work is supported by National Natural Science Foundation of China under Grant No. 61972376, No. 62072431, No. 62032023.

References

[1] H. Jia, Y. Zhang, G. Long, et al. "GPURoofline: A Model for Guiding Performance Optimizations on GPUs," Proc. 18th Int'l Conf. Parallel Processing (Euro-Par '12), vol. 7484, pp. 920- 932, 2012.
[2] K.Group,"OpenCL." http://www.khronos.org/opencl(accessed Jul. 26, 2022).
[3] Nvidia,"CUDA." https://docs.nvidia.com/cuda/(accessed Jul. 26, 2022).
[4] K. Group,"OpenGL." http://www.khronos.org/opengl(accessed Jul. 26, 2022).
[5] D. Wolffff, OpenGL 4 Shading Language Cookbook, 2nd ed. Packt Publishing Ltd., December 2013.
[6] R. Thabet, R. Mahmoudi and M. H. Bedoui, "Image processing on mobile devices: An overview," International Image Processing, Applications and Systems Conference, 2014, pp. 1-8, doi: 10.1109/IPAS.2014.7043267.
[7] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: An opensource gpu-accelerated framework for image processing and computer vision," in Proc. of the 16th ACM international conference on Multimedia, October 2008, pp. 1089– 1092.
[8] P. Babenko and M. Shah, "MinGPU: A minimum gpu library for computer vision," Real-Time Image Processing, vol. 3, no. 4, pp. 255–268, 2008.
[9] J. Fung, S. Mann, and C. Aimone, "OpenVIDIA: Parallel gpu computer vision," in Proc. of the 13th annual ACM international conference on Multimedia, November 2005, pp. 849–852.
[10] N. Singhal, I. K. Park and S. Cho, "Implementation and optimization of image processing algorithms on handheld GPU," 2010 IEEE International Conference on Image Processing, 2010, pp. 4481-4484, doi: 10.1109/ICIP.2010.5651740.
[11] G. Bradski, "Open source computer vision library." https://docs.opencv.org/4.6.0/(accessed Jul. 28, 2022).
[12] Wijayanti, Sari et al. "A New Native Video Filtering based on OpenGL ES for Mobile Platform." International Journal on Advanced Science, Engineering and Information Technology (2019): n. pag.
[13] C.-H. Chou, P. Liu, T. Wu, Y. Chien, and Y. Zhao, "Implementation of parallel computing fast algorithm on mobile gpu," in Unifying Electrical Engineering and Electronics Engineering, pp. 1275–1281, Springer, 2014.
[14] B. Rister, G. Wang, M. Wu, and J. R. Cavallaro, "A fast and efficient sift detector using the mobile gpu," in Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pp. 2674–2678, IEEE, 2013.
[15] K.-T. Cheng and Y.-C. Wang, "Using mobile gpu for general-purpose computing–a case study of face recognition on smartphones," in VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on, pp. 1–4, IEEE, 2011.
[16] "ARM MALI Visual Technology OpenGL ES SDK for Android: Introduction to compute shaders." https://arm-software.github.io/opengl-es-sdk-for-android/compute_intro.html(accessed Jul. 26, 2022).
[17] H. Yeganeh, A. Ziaei and A. Rezaie, "A novel approach for contrast enhancement based on Histogram Equalization," 2008 International Conference on Computer and Communication Engineering, 2008, pp. 256-260, doi: 10.1109/ICCCE.2008.4580607.
[18] A. de la Torre, A. M. Peinado, J. C. Segura, J. L. Perez-Cordoba, M. C. Benitez, and A. J. Rubio, "Histogram equalization of speech representation for robust speech recognition," IEEE Trans. Speech Audio Processing, vol. 13, pp. 355-366, May 2005.
[19] Lindeberg, Tony. Scale-space theory in computer vision. Vol. 256. Springer Science Business Media, 2013.
[20] Dang, Qingqing, Shengen Yan, and Ren Wu. "A fast integral image generation algorithm on GPUs." 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2014.
[21] P. Viola and M. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features", Conference on Computer Vision and Pattern Recognition (CVPR), pp.511-518, 2001.
[22] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column-stores. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09). Association for Computing Machinery, New York, NY, USA, 297–308.
[23] X.An, Y.Zhang and H.Jia. Research on Histogram Generation Algorithm Optimization Based on OpenCL[J]. Computer Science, 2015, 42(11): 32-36.
[24] Bilgic B, Horn B K P, Masaki I. Efficient integral image computation on the GPU[C]//2010 IEEE Intelligent Vehicles Symposium. IEEE, 2010: 528-533.
[25] H.Jia.Research of Parallel Optimization Technicals on GPU Computing Platforms.2012.Ocean University of China,PhD dissertation.
[26] H.Jia, Y.Zhang, G.Long, et al. An insightful program performance tuning chain for GPU computing[C]//International Conference on Algorithms and Architectures for Parallel Processing. Springer, Berlin, Heidelberg, 2012: 502-516.