GPGPU 기반의 효율적인 카메라 ISP 구현

박종태*, Beorn Facchini*, 홍진건*, Bernd Burgstaller*
*연세대학교 컴퓨터과학과
e-mail : {jongtae.park, beorn, ginug, bburg}@cs.yonsei.ac.kr

Implementing Efficient Camera ISP Filters on GPGPUs Using OpenCL

Jongtae Park*, Beron Facchini*, Jingun Hong*, Bernd Burgstaller*
*Dept. of Computer Science, Yonsei University

요 약

General Purpose Graphic Processing Unit (GPGPU) computing is a technique that utilizes the high-performance many-core processors of high-end graphic cards for general-purpose computations such as 3D graphics, video/image processing, computer vision, scientific computing, HPC and many more. GPGPUs offer a vast amount of raw computing power, but programming is extremely challenging because of hardware idiosyncrasies. The open computing language (OpenCL) has been proposed as a vendor-independent GPGPU programming interface. OpenCL is very close to the hardware and thus does little to increase GPGPU programmability. In this paper we present how a set of digital camera image signal processing (ISP) filters can be realized efficiently on GPGPUs using OpenCL. Although we found ISP filters to be memory-bound computations, our GPGPU implementations achieve speedups of up to a factor of 64.8 over their sequential counterparts. On GPGPUs, our proposed optimizations achieved speedups between 145% and 275% over their baseline GPGPU implementations. Our experiments have been conducted on a Geforce GTX 275; because of OpenCL we expect our optimizations to be applicable to other architectures as well.

1. Introduction

Researchers and developers alike have become interested in accelerators for extremely demanding computations. As a result, GPUs, which have been originally intended for 3D graphics, have recently acquired general purpose computing capabilities to exploit the abundant amount of parallelism offered by these architectures. This trend is collectively known as GPGPU computing [5]. In response to increasing numbers of vendor-specific platforms and programming interfaces, OpenCL, has been proposed as an open standard for GPGPU computing. Digital image processing is an application domain with an abundant amount of parallelism: algorithms often apply to single pixels or sub-parts of an image, with no (or little) dependencies between them. Nevertheless, because of GPGPU hardware idiosyncrasies, achieving high performance for such algorithms is still a challenging task. In this paper we investigated how camera ISP filters can be implemented on NVIDIA's CUDA architecture using OpenCL. Our work shows how performance bottlenecks with ISP algorithms can be avoided and how programming techniques for GPGPUs can be utilized specifically for image processing. The remainder of the paper is structured as follows: in Section 2 we survey related work. In Section 3 we introduce our filter algorithms and the GPGPU-specific optimizations. In Section 4 we present our experimental results. We draw our conclusions in Section 5.

2. Related Work

2D image processing has been achieved through a restricted form of 3D graphics processing since GPUs have become programmable. Early approaches required knowledge of GPU programming like Direct3D or OpenGL in which each pixel can run one or more small programs called vertex shaders and pixel shaders. Mcguire's demosaicing and median filters show reasonable performance on GPUs with this approach [4, 5].
In this paper we approach image processing as a form of GPGPU computing based on OpenCL. OpenCL is geared towards general-purpose computing and it is platform-independent, which will provide a straight-forward mapping (i.e., re-compile) of code onto other types of parallel accelerators such as the IBM Cell Broadband Engine or DSPs.

3. Camera ISP Filters

Many different image filter algorithms exist, but very few are actually required for camera ISP pipelines. We focused on common filters, including filters from the post-ISP pipeline, for which we were able to derive optimization methodologies that will apply to the majority of image filter algorithms. Our implementations comprise five kinds of filters: color reconstruction (a.k.a. demosaicing), color model conversion, noise reduction, image convolution and pixel blending. Although pixel blending is not an essential component of ISP pipelines, it is a computationally intensive filter. Mapping such a filter on a GPGPU can yield reasonable frame rates.

3.1 Work-group Size

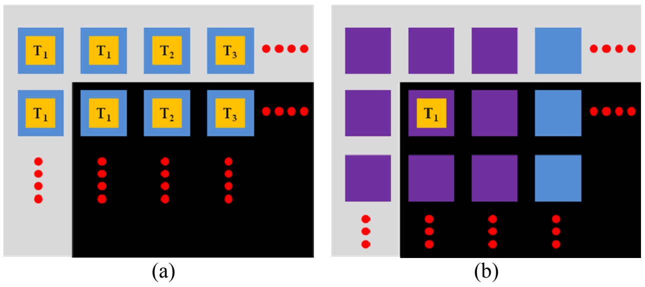
The work-group size is a crucial factor that determines how effectively the GPGPU CUs are utilized. Note that this factor depends on how many active threads and work-groups a CU on a particular GPGPU can have. In other words, it depends on the hardware specification. Moreover, even if we find best block size, we cannot use it when an input image is not divisible by that size. It is therefore hard to make a decision of a proper block size. Nonetheless, we could achieve reasonable performance by following a basic idea in which we schedule threads as many as possible onto CU. In this paper, we employed a 16x20 work-group which results in 93.8% utilization of feasible threads of NVIDIA GTX275 CUs (total work-groups: 3, total scheduled threads: 960, the maximum number of active threads per CU: 1024). We could attain speedups over arbitrary work-group sizes in all filters using this size and it works best in our loading scheme (see below). A block size divisible by 32 usually shows good performance because CUDA schedules 32 threads (called warp) at once.

3.2 Utilizing Local Memory

It is common with image processing algorithms that the computation for a given pixel refers to neighboring pixels (e.g., with mask-based algorithms like thresholding or averaging, see also Fig. 1(b)). Overall, such algorithms read each pixel value several times, which suggests to load pixel data from global memory into local memory; local memory is a CU-specific low-latency memory [2] with about 100x lower memory latency than global memory. The more neighboring pixels on the global memory we refer to, the bigger performance degradation we experience, especially with large-sized masks. It is therefore important to utilize local memory to hide the latency to global memory for performance gain. For this, the basic idea is that we load work-items from the global memory and store these into the local memory only once. After that, we never refer to work-items that reside in global memory during the filter invocation. Instead, we refer to the work-items on the local memory as many times as we want. Nevertheless, It will barely causes the performance degradation thanks to the lower memory latency, as mentioned above.

3.3 Loading Scheme for Global Work-items

A straight-forward way to load data into local memory is to have each thread load one data-item. This scheme is optimal, but it is not applicable with filters that involve aprons, e.g., with convolution filters, because we need to load the pixels in the apron region as well. Fig. 1(a) depicts a loading scheme of apron pixels (gray area) by threads $T_1 \sim T_3$ and Fig. 1(b) depicts thread T1 applying a 3x3 mask (purple-colored) on the input pixel marked 'T_1'. A naïve way to load apron pixels is to have one thread of each work-group load all work-items into local memory. We define this as a *one thread loading scheme*. But it obviously introduces a bottleneck that makes all other threads wait for the 'loading thread' to complete memory transfers of apron pixels.



(Figure 1) Loading work-items and applying a 3x3 mask

An obvious improvement will be to have multiple threads of a workgroup share the loading of apron pixels. As indicated in Fig. 1(a), a loading scheme that assigns each apron pixel to a thread needs to be devised. This loading scheme needs to balance the load among threads equally, and it needs to minimize executions of diverging branches: branch divergence occurs if threads of the same warp take different branches (with control-flow statements, e.g., if, switch, for, while). If this happens, the different execution paths must be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads of the warp continue in data-parallel way along a single execution path. In this paper, we propose the following loading scheme that shows a 55.8 speedup over the one thread loading scheme (for a 9x9 mask).

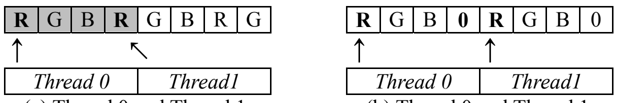
```
1:  $x\_offset \leftarrow globalThreadID.x - (localThreadID.x + filterMaskSize)$
2:  $y\_offset \leftarrow globalThreadID.y - (localThreadID.y + filterMaskSize)$
3:  $y \leftarrow localThreadID.y * 2$
4:  if $y < localMemoryHeight$ then
5:      $x \leftarrow localThreadID.x$
6:      $l\_offset \leftarrow localMemWidth * y + x$
7:      $g\_offset \leftarrow imageWidth * (y + y\_offset) + x + x\_offset$
8:      $localMem[l\_offset] \leftarrow globalMem[g\_offset]$
9:      $localMem[l\_offset + localMemoryWidth] \leftarrow globalMem[g\_offset + imageWidth]$
10:     if $x < maskSize * 2$ then
11:         $lhsOffset \leftarrow l\_offset + workGroupWidth$
12:         $rhsOffset \leftarrow g\_offset + workGroupWidth$
13:         $localMem[lhsOffset] \leftarrow globalMem[rhsOffset]$
14:         $lhsOffset \leftarrow l\_offset + localMemoryWidth + workGroupWidth$
15:         $rhsOffset \leftarrow g\_offset + imageWidth + workGroupWidth$
16:         $localMem[lhsOffset] \leftarrow globalMem[rhsOffset]$
17:     end if
18: end if
```

(Figure 2) Pseudo code for proposed loading scheme

In our loading scheme, each row of threads loads two pixel-rows from the global memory and some of threads in a row also participate in loading apron pixels. Lines 11 ~ 19 take into account this operation. Although some rows in a work-group are idle (e.g., 4 out of 16 for 16x20 work-groups) this scheme is much faster than the one thread loading scheme and even two times faster than the loading scheme recommended by NVIDIA in the CUDA 3.0 toolkit, because of the row by row loading scheme without concern for top/bottom apron pixels. The number of diverging branches is 1512 for our loading scheme and 4344 for NVIDIA's loading scheme. Note that our proposed loading scheme can be applied to any mask size as long as the width and height of a work-group are at least two times bigger than the radius of a mask. In contrast, NVIDIA's proposed loading scheme only supports a mask size of 3x3 pixels.

3.4 Padding Local Memory

Local memory consists of memory blocks called *banks*, each of which has 32 bit words. If N memory requests from executing threads access the same memory bank, a *bank conflict* will arise, increasing N times the latency of local memory access. In the RGB color model 24 bits color arrays cause bank conflicts as depicted in Fig. 3(a).



(a) Thread 0 and Thread 1 access the same bank

(b) Thread 0 and Thread 1 access different banks

(Figure 3) Memory bank accesses of thread 0 and 1

To avoid this, we padded 8 bits at the end of each pixel so that a thread's request for a pixel's values (R,G,B) is restricted to a single discrete memory bank. Fig. 3(b) shows how we pad a RGB array with 8 bits. Padding improved performance of RGB-based filters by 2% - 8%.

3.5 Predicated Execution

One feasible way to reduce the number of branches is to use predicated instructions. With predicated instructions the controlling condition (e.g., from an if statement) is attached to the instruction itself. At run-time, the instruction is scheduled for execution, but only executed if the condition is true. This is called *predicated execution* [2]. If we write a program in a way such that conditional constructs have two branches, then the OpenCL compiler schedules both branches using predicated instructions (if the number of statements controlled by the condition is below a given threshold).

```
1: offset ← width * y + x
2: if  x > 1000  then
3:    g_out[offset] ← 0
4: else then
5:    g_out[offset] ← 255
6: end if
```

(a) a non-optimized conditional construct example

```
1: offset ← width * y + x
2: if  x > 1000  then
3:    out ← 0
4: else then
5:    out ← 255
6: end if
7: g_out[offset] ← out
```

(b) an optimized conditional construct example

```
1: coord ← ( x, y ) mod ( 2, 2)
2: if  coord.y = 1  then
3:    if  coord.x = 1  then
4:       out ← mask[0];
5:    else then
6:       out ← mask[1]
7:    end if
8: else if  coord.y = 0  then
9:    if  coord.x = 1  then
10:      out ← mask[2];
11:   else then
12:      out ← mask[3]
13:   end if
14: end if
```

(c) optimized conditional constructs of demosaicing

(Figure 4) non-optimized/optimized conditional constructs

We now consider applying predicated execution to demosaicing. Based on the coordinates (x, y), there are four possible conditions in the Bayer-patterned input image [4], see also Fig. 4(c). A combination of "if", "else if" and "else" or "switch ~ case" statements can be used for a thread to decide which condition applies. Let's assume that we use a combination of one "*if*", two "*else if*" and one "*else*" for demosaicing. In this case, we will have at least two branch points and one convergence point. Because the computation of the target address for the indirect memory address (lines 3 and 5) requires enough instructions to exceed the compiler's threshold for branch predication, a branch is introduced. (Note that the threshold for the number of instructions for predicated execution is 3 or 6 in CUDA [2].) We can stay below the threshold by introducing a temporary variable for *g_out*, which doesn't include address calculations for the indirect memory access. The optimized code that uses predicated execution is depicted in Fig. 4(b). The code for all four cases is shown in Fig. 4(c). Note that, although we removed just one branch by this schedule, we could acquire a 29% performance increase by this optimization.

For median filter, sorting algorithms can be commonly used for median filtering and we chose *insert sort* as the baseline. To apply predicated execution, a proper approach is Paeth's algorithm [6] in which the minimum and the maximum of the first six elements in an array are determined and eliminated without applying a loop. Then the 7th, 8th, 9th elements are moved to the empty space and repeated previous actions until the median is determined (3x3 mask). We can change all comparisons in Paeth's median filter algorithm into two-directional conditional constructs similar to what McGuire showed in [4, 5], because only comparisons of two values and swap instructions are involved in this algorithm. It means we can also apply predicated execution to the median filtering easily, eliminating all possible branches. With our predicated execution the number of diverging branches was reduced from 7866 to 3832 for the demosaicing filter (by 51%), and from 73679 to 2059 for median filter (by 97%).

3.6 Pinned Memory

Host memory paged out to the hard-disk can affect program performance due to lower memory bandwidth. However, OpenCL provides an API to lock data into the host's memory (*pinned memory* or *page-locked memory* [1]). Therefore, pinned memory use is useful only when data is being transferred between the host and the device. However, it should be noted that pinned memory might cause performance degradation of other programs running on the machine as there would be a higher chance for those programs to be paged-out from physical memory.

4. Experimental Results

To evaluate our proposed optimizations, we implemented filters using the CUDA SDK 3.0 ToolKit on CentOS 5.5 and a Geforce GTX 275 equipped with 30 CUs and 1 GB of device memory. In addition to GPGPU filters, we implemented the same filters for single thread execution, on an Intel quad-core i5 750 2.67MHz to determine the speedups of parallelized filters over sequential execution. In the following, all execution times are given in milliseconds, and the resolution of input images for Sections 4.1 and 4.2 is 1920x1080, full HD. To test the median filter, we added 10% of salt-and pepper noise before running the filter but the time of adding noise is not added to the actual time for measurement.

<Table 1> Cumulative results from left to right column after a series of optimizations (*:Maximum-optimized result)

Filter Type	CPU Sequential	Global 4x4	Global 16x20	Global 16x20	Local 16x20	Local Pad. 16x20	Local Pad. 16x20	CPU/GPU Speedup	Base/Opt. Speedup
		Baseline	Best Size	Pinned	Local Mem.	Padding	Predicated Exec.		
Blur 3x3	147.4	8.00	6.75	6.24	5.94	***5.52**	N/A	26.7	1.45
Sharp 3x3	150	8.00	6.78	6.27	5.98	***5.57**	N/A	26.9	1.44
Median 3x3	666	21.38	15.86	15.34	15.02	14.52	***10.33**	**64.8**	2.06
Demosaicing 5x5	117.92	13.53	11.24	7.95	6.66	6.46	***4.92**	23.9	**2.75**
Blur 5x5	332.4	16.20s	13.21	12.68	9.46	***8.88**	N/A	37.4	1.82
Sharp 5x5	333.2	16.22	13.37	12.82	9.49	***8.89**	N/A	37.5	1.82
Median 5x5	2700	119.75	81.80	81.26	79.07	78.23	***44.45**	61.1	2.69
Oil 9x9	2325	237.5	240.6	240.1	211.9	***167.5**	N/A	13.9	1.42

4.1 Comparison of Loading Schemes

Table 2 shows the execution times for four approaches of loading all pixels in the 16x20 work-group into local memory. "Self" is the simplest as each thread loads one corresponding global memory block. "One thread" means that only one thread loads all work-items into local memory and "NVIDIA" means the loading scheme used in the sample of NVIDIA CUDA SDK 3.0 Toolkit. You can see how significantly idle threads caused by one thread scheme deteriorate the loading time.

<Table2> four approaches of loading global memory into local memory

Mask size	Self	Our scheme	NVIDIA	One thread
3x3	3.47	3.74	4.26	22.49
5x5	3.47	3.74	N/A	26.49
7x7	3.47	3.81	N/A	30.88
9x9	3.47	3.81	N/A	35.64

All loading schemes here include global memory access and it always takes 3.23 ms regardless of mask size. Therefore, you can think that our scheme is actually two times faster than NVIDIA scheme if we subtract 3.23 ms from measured times

4.2 Cumulative Results after Optimizations

Table 1 presents cumulative results after a series of optimizations proposed in this paper. The second column shows the execution times of image filter programs for sequential execution and the third column indicates the baseline implementation for our optimizations. Actual execution times after each optimization begin from 4th column towards the right and indicating proper block size, pinned memory use, local memory use, padding RGB and predicated execution, respectively. The 8th column shows speedups over all optimizations. We achieved from 1.42 to 2.75 speedups over the GPGPU baseline and a maximum speedup of 64.8 over the sequential case. Note that filters with bigger mask size show more performance gain from local memory use except for the median filter. The reason for low performance gains of median can be explained by the fact that the median filter stores again work-items into an array for sorting them. But this overhead can be naturally amortized by predicated execution as can be inferred from the 8th column.

4.3 Consistency of Scalability for Work-items

Many-core accelerators such as GPGPU should show consistent performance results based on the number of input work-items in terms of scalability. Thus we conducted experiments to verify the consistency of scalability for the number of work-items which is equal to the number of pixels in the input image. As you can see at Table 3, Full-optimized median filter still show the same scalability before being optimized. Speedup here means the speedup over the previous smaller input image.

<Table 3> Execution time for the number of work-items (resolutions) for median filter with 3x3 mask

	1920x1080	1280x720	720x480	640x480
Baseline	21.38	9.72	3.87	3.48
Speedup over prev.	2.19	2.51	1.11	-
Full-optimized	10.33	4.58	1.80	1.60
Speedup over prev.	2.25	2.54	1.12	-

5. Conclusion

We have presented a set of optimizations for camera ISP filters, taking various hardware features of GPGPUs into account. Our filter algorithms have been implemented in OpenCL, which is a vendor-independent programming interface for GPGPUs. We expect our algorithms to show similar results when realized on hand-held embedded devices such as tablet PCs and smart-phones.

References

[1] OpenCL Overview. URL http://www.khronos.org/developers/library/overview/opencl_overview.pdf

[2] NVIDIA OpenCL Programming Guide. URL http://www.nvidia.com/OpenCL

[3] M. McGuire. A fast, small-radius gpu median lter. In ShaderX6, 2008.

[4] M. McGuire. Efficient, high-quality bayer demosaic Fltering on GPUs. J. Graphics Tools, 13(4):1-16, 2008.

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26(1):80-113, 2007.

[6] A. W. Paeth. Median finding on a 3x3 Grid. In Andrew Glassner, editor, Graphic Gems, pages 171-175. Academic Press, Boston, 1990.