# An Optimization Scheme for Demosaicing Algorithm on GPU Using OpenCL

Tongli Wang, Wei Guo, and Jizeng Wei(✉)

School of Computer Science and Technology, Tianjin University, Tianjin, China
{tongliwang,weiguo,jizengwei}@tju.edu.com

**Abstract.** With the popularity of GPU which has the high performance computing feature, more and more algorithms have been successfully transplanted to the GPU platform and achieved high efficiency. But existing videos or images processing methods, such as demosaicing algorithm, have not fully exploited the parallel computing capacity of heterogeneous processing platform and the video frame rates can't meet real-time requirements. In order to take full advantage of the computing power of GPU under the heterogeneous processing platform, an optimization scheme is proposed in this paper. We use the demosiacing algorithm as a case and modify the algorithm. By exploiting the GPU's memory hierarchy, the optimization scheme improves the parallelism of the algorithm while reducing the memory access latency, and greatly reduces the execution time. Then we achieve the zero-copy at the same time. The experimental results show that optimization version has a significant performance improvement, the optimized OpenCL version is up to 6x comparing with the basic OpenCL version about kernel execution.

**Keywords:** Parallel processing · Image demosaicing
Heterogeneous platform · OpenCL

## 1 Introduction

Digital cameras are increasingly widespread, and camera modules are now embedded in a variety of handheld devices including mobile phones and tablet PCs. Due to the cost of imaging, most digital camera imaging chips only have one CMOS or CCD sensor chip, each sensor surface is covered with a color filter array [1,2] (Color Filter Array, CFA), such as Fig. 1. The conventional color filter array limits the arrival of only one base light per pixel location, capture only one color component at each spatial location. The remaining components must be reconstructed by interpolation from the captured samples. So that the other two colors of the color image will be interpolated with the sampling result of the adjacent pixels of the sampling matrix in the case of single block inductive chip samples [3]. This color plane interpolation algorithm is called image to mosaic. In the early stage of the computer technology, graphics processing and computing are relatively simple, we can use the CPU to achieve graphics processing.

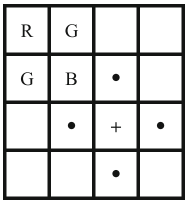

**Fig. 1.** Bayer CFA

But with the development of computer technology, especially the requirements on the quality of graphics processing and computing speed continue to improve, this needs to find new ways to meet the increasing requirements.

Nowadays, parallel computers are not expensive and exquisite because almost all PCs have multicore hardware. Basically, there are two main multi-core approaches: integrating some of the core into a single microprocessor (multi-core CPU), or integrating a large number of cores to the current graphics processing unit (GPU) as an example [4]. The GPU was originally designed specifically for graphics applications and image rendering required during the rasterization process. Over time the computational resources of modern graphics processing units became suitable for certain general parallel computations because of the inherent parallel processing capabilities of the architecture [5]. By starting multiple execution threads, we can take advantage of all of these multicore hardware.

So, heterogeneous computing of CPU and GPU become the mainstream platform of high performance computing, which has great advantages in computing energy efficiency compared with multi-core processors and has been well verified by the parallelization of multiple algorithms.

In this paper, we propose an optimization scheme for demosaicing algorithm. The objective of this implementation is to demosaic image as fast as possible, so that the video editing workflow will be accelerated. To achieve this, we first introduce the parallel processing of the algorithm as the base method. Then we propose two implementation methods, one is reducing input and output transfer between global and shared memory when data transmission between GPU and CPU, another is reducing the number of work items and queuing time by changing the distribution of working groups. Finally, we come to the conclusion.

## 2 Related Work

[6] proposed an improved linear interpolation for demosaicking of Bayer-patterned color filter array (CFA) images. An efficient edge-based technique for color filter array demosaicking is presented in [2]. The authors in [1] introduce an efficient demosaicking method based on an advanced nonlocal mean filter using adaptive weight with consideration of both neighborhood similarity and patch distance.

Meanwhile, several works have been dedicated to implement demosaicing using GPU. An efficient implementation of Bayer demosaic filtering on GPUs was published in [7]. McGuire accelerated MalvarHe-Cutlere [8] image demosaicing algorithm using OpenGL in real-time speed.

OpenCL is the first open, free standard for parallel programming for general purpose heterogeneous systems and a unified programming environment, which is used to program multiple devices, including GPU and CPU, as well as other computing devices as part of a single computing platform. OpenCL uses parallel execution SIMD (single instruction, multiple data) engines found in General Purpose Graphics Processing Units (GPGPU) and Compute Cores(CC) to enhance data computational density by performing massively parallel data processing on multiple data items, across multiple compute engines. Each compute unit has its own ALUs, including pipelined floating-point (FP) units, integer (INT) units that can perform computations as well as transcendental operations.

Due to the good cross platform and parallelism of OpenCL, in recent years, OpenCL has also been widely used in image processing and algorithm acceleration. For example, [5] proposes a parallel implementation and optimization method for the real-time dehazing of the high definition videos based on a single image haze removal algorithm.

In this paper, we further modified and optimized the demosaicing algorithm. The presented OpenCL implementation in paper is 6 times faster than the GPU implementation in [7] using the same filter. And we use the 4th Generation Intel® Core$^{TM}$ Processor family which includes complex SoCs integrating multiple CPU Cores, Intel® Processor Graphics, and potentially other fixed functions all on a single shared silicon die. And the GPU and CPU share the Last Level Cache (LLC).

## 3 Parallel Implementation and Optimization of Demosiacing Algorithm Based on OpenCL

In an OpenCL execution model, the host program is responsible for managing and scheduling OpenCL-supported computing devices. When the host side submits the kernel to computing devices, serial code defines the organization structure of the work item through the global index space (NDRange) and the operation mode of the kernel on the computing device through the mapping method on the computing device, as shown in Fig. 2.

Figure 3 shows that the OpenCL memory architecture is divided into four parts: global memory, constant memory, local memory, and private memory, as shown in the figure. The sizes and corresponding access speeds of these memory types are different. Data can flow along the channel of host memory, global memory, local memory, private memory. When optimizing the OpenCL kernel program, it‘s an important part to fully tap the potential of the GPU's storage hierarchy based on the characteristics of the algorithm.

### 3.1 Algorithm Modification

The Intel Graphics device is equipped with several Execution Units (EUs). EUs are Simultaneous Multi-Threading (SMT) compute processors that drive

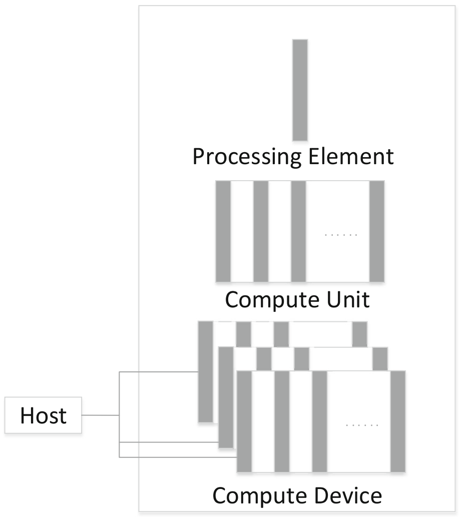

**Fig. 2.** OpenCL platform model

multiple issuing of the Single Instruction Multiple Data Arithmetic Logic Units (SIMD). Compiler generates SIMD code to map several work-items to be executed simultaneously within a given hardware thread. The SIMD-width for kernel is a heuristic driven compiler choice. Therefore, the basic algorithm version suffers a significant performance improvement.

[6] presented an OpenGL implementation of the Malvar-HeCutler filter. [7] also provide a GPU Filters which includes the filter coefficients. And the GPU Filters can achieve SIMD such as MADD and ADD on 4-vectors at the same speed as on scalars. For example, when calculating the float4 value PATTERN, we use the following formula:

$$\begin{aligned} PATTERN+ =& (kA.xyz * (float3)(value.x, value.x, value.x)).xyzx+ \\ & (kE.xyw * (float3)(value.z, value.z, value.z)).xyxz \end{aligned} \tag{1}$$

There are many similar formulas in the kernel to adapt to the characteristics of SIMD. This will make the most advantage of SIMD and reduce the amount of calculation steps and running time.

For a given SIMD-width, if all kernel instances within a thread are executing the same instruction [12], then the SIMD lanes can be maximally utilized. Moreover, the GPU instruction execution is SIMD, the GPU Vector ALU hardware is more flexible and can efficiently use the floating-point hardware [13]. In this paper, we modified the algorithm code, a lot of uchar8 and float8 data types are used to further speed up the program running time, including addition, multiplication, dot times and other operations. So we can make full use of the SIMD-width. For example:

$$uchar8\ lineA = (uchar8)(vload8(0, psrc + mad24(j-2, 1920, i*4-2))) \tag{2}$$

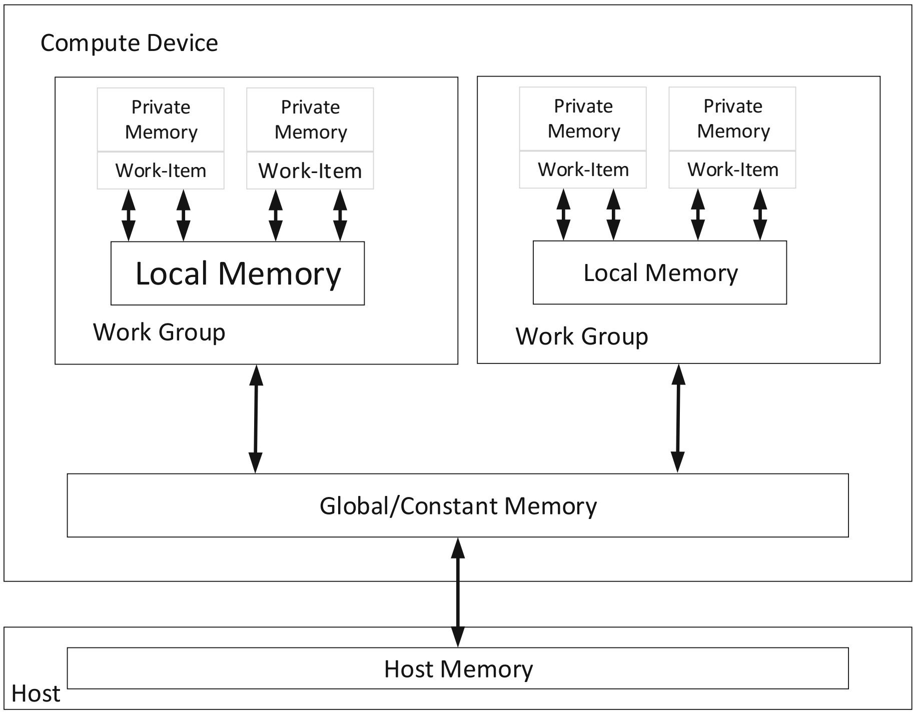

**Fig. 3.** OpenCL memory architecture

$$\begin{aligned}out =&(uchar16)(lineC.s2, convert_uchar2(PATTERN\_One.xy), 255,\\ &PATTERN\_Two.z, lineC.s3, PATTERN_Two.w, 255,\\ &lineC.s4, convert_uchar2(PATTERN\_Three.xy), 255,\\ &PATTERN\_Four.z, lineC.s5, PATTERN\_Four.w, 255);\end{aligned} \tag{3}$$

Due to the SIMD-width is fully occupied when operations execute, an obvious performance improvement when executed on GPU environment [12]. In addition, by doing so, we can handle four pixels at a time. Algorithm 1 shows the steps of the modified version.

### 3.2 Data Transmission Optimization

When mapping OpenCL on CPUs, the host and device share the same memory space [4]. Since OpenCL requires explicit data transfers but does not impose restrictions on memory access patterns, it is up to the compiler and to the device driver to select whether or not to actually replicate the data or just read it from already allocated space, and Fig. 4 is the traditional mode of data transmission. To overcome this irregularity, we applied the so called zero copy technique.

To achieve zero copy, the Intel Processor Graphics has a congenital advantage. Intel® Processor Graphics architecture shares DRAM physical memory

**Algorithm 1.** Optimization demosiacing algorithm

```
Input: input A 8-bit gray 1920 × 1080 image
Output: output A 32-bit color 1920 × 1080 image
1: Using vload8 instruction to obtain the pixels and assigned to lineA˜ lineE
2: Calculate the filter coefficients
3: Calculate the pattern 4-vector of filter terms
4: Using the pattern to restore the four color pixels A,B,C,D
5: return out = (uchar16)(A,B,C,D)
```

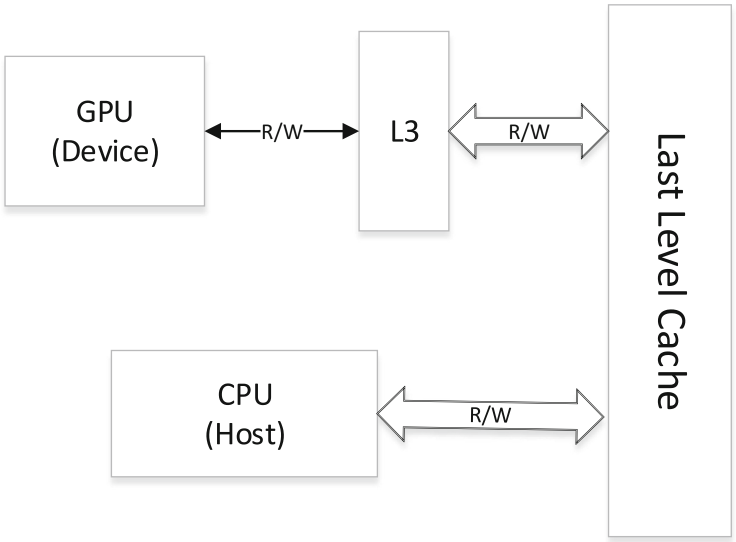

**Fig. 4.** The original data transfer method

with the CPU like Fig. 5. Thus, the advantage is that shared physical memory enables zero-copy buffer transfers between CPUs and Gen7.5 compute architecture. Moreover, the architecture further augments the performance of this sharing with shared caches. This reduces the overhead of the data transfer.

All data into and out of the samplers and data ports flows through the L3 data cache in units of 64-byte wide cachelines. This includes read and write actions on general purpose buffers. L3 cache bandwidth efficiency is highest for read/write accesses that are cacheline aligned and adjacent within cacheline. Compute kernel instructions that miss the subslice instruction caches also flow through the L3 cache. A kernel should access at least 32-bits of data at a time, from addresses that are aligned to 32-bit boundaries.

In order to improve performance, we use the vload8 and vstore8 to read data from shared memory. On one hand, this will reduce the data transfer time. On the other hand, this also allows four pixels are restored at one time in kernel like Fig. 6.

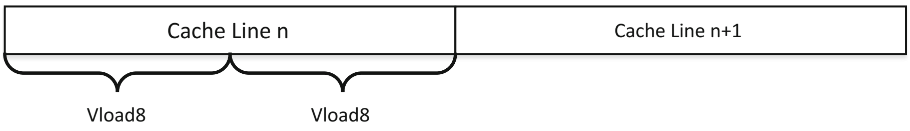

**Fig. 5.** The optimized data transfer mode

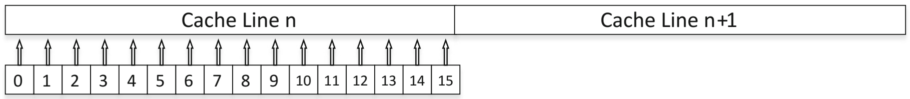

**Fig. 6.** Vload data to cache at once

### 3.3 Memory Management and Indexes Memory

There are global memory and local memory in INTEL® PROCESSOR GRAPHICS. How to manage the memory will influence the data progress. In this paper, we use a 1920 × 1080 image as an example.

In general, we will allot the size of the image as global memory. Because our kernel will use vload8 to read data, this will waste the memory. So we can shrank a quarter in size and shorten the time about a half. Further, we can set the local memory a multiple of 32, which is the SIMD-width. This is because the work-item will share the local memory, and a SIMD-width size can be suitable the data width.

To optimize performance when accessing __global memory, a kernel must minimize the number of cache lines that are accessed [11]. If a kernel indexes memory, where index is a function of a work-item global id(s), the following factors have big impact on performance:

i The work-group dimensions
ii The function of the work-item global id(s)

The work-group dimensions can affect memory bandwidth. We call a "row" work-group: <16, 1, 1>. With the "row" work-group, get_global_id(1) is constant for all work-items in the work-group, myIndex increases monotonically across the entire work-group, which means that the read from, and myArray comes from a single L3 cache line (16 x sizeof(int) = 64 bytes) like Fig. 7. This will make full use of the bandwidth to read data from cache line.

Also, the function of the work-item global ids can affect memory bandwidth [11]. In our kernel, we use the following way to get work-item ids.

$$\begin{aligned} &int\ i = get\_global\_id(0);\\ &int\ j = get\_global\_id(1);\\ &int\ src\_idx = mad24(j, 1920, i * 4);\\ &int\ x = psrc[src\_idx]; \end{aligned} \tag{4}$$

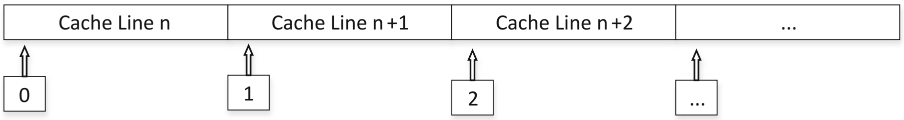

**Fig. 7.** The read is cache-aligned, and the entire read comes from one cache line. This case should achieve full memory bandwidth

The read is cache-aligned, and the entire read comes from one cache line. This case should achieve full memory bandwidth. This will get full the memory performance. The read from psrc comes from same L3 cache line for 16 work-items. This means a single L3 cache line (16 x sizeof(int) = 64 bytes) will full used.

## 4 Experimental Results and Performance

We implement the demosiacing algorithm by three ways. First, we use a straightforward CPU implementation with the filter in [8] using C++ programming. Second, we first implement the basic OpenCL version using the GPU filter in [7]. And the final implementation is the optimized OpenCL version. And the time are divided into three parts: write data to device, read Data from device and kernel execution.

The tests reported in this study were performed on a multiprocessor PC with an Intel(R) HD Graphics 4600 and an Intel core i7-4590 3.30 GHz CPU. Each CPU of the pc has 4 physical cores. As each physical core hosts two virtual cores. The C++ development environment is Microsoft Visual Studio 2017. The OpenCL development environment is an intel sdk with OpenCL version 1.2.

In our paper, we use the 8-bit gray images of three size including $640 \times 480$, $1024 \times 768$ and $1920 \times 1080$. To evaluate the performance on GPU, all versions were run 50 times. Table 1 shows the execution times.

**Table 1.** Execution times for three image sizes

| Image size | Version | Write data (ms) | Read data (ms) | Kernel execution (ms) |
|---|---|---|---|---|
| $640 \times 480$ | CPU version | NULL | NULL | 52.8441 |
| $1024 \times 768$ | | NULL | NULL | 81.4833 |
| $1920 \times 1080$ | | NULL | NULL | 131.3766 |
| $640 \times 480$ | Basic OpenCL | 0.6958 | 0.8429 | 0.7288 |
| $1024 \times 768$ | | 0.8958 | 1.3232 | 1.0784 |
| $1920 \times 1080$ | | 1.2533 | 2.0968 | 1.9415 |
| $640 \times 480$ | Optimized OpenCL | 0.0154 | 0.0123 | 0.2876 |
| $1024 \times 768$ | | 0.0165 | 0.0133 | 0.4249 |
| $1920 \times 1080$ | | 0.0167 | 0.0143 | 0.7473 |

**Table 2.** Execution times for three platforms

| GPU type | Version | Write data (ms) | Read data (ms) | Kernel execution (ms) |
|---|---|---|---|---|
| HD4600 | CPU version | NULL | NULL | 131.3766 |
| HD530 | | NULL | NULL | 108.4523 |
| HD630 | | NULL | NULL | 90.5148 |
| HD4600 | Basic OpenCL | 1.2533 | 2.0968 | 1.9415 |
| HD530 | | 1.0542 | 1.3376 | 1.1365 |
| HD630 | | 0.9856 | 1.1232 | 0.9147 |
| HD4600 | Optimized OpenCL | 0.0167 | 0.0143 | 0.7473 |
| HD530 | | 0.0163 | 0.0145 | 0.5173 |
| HD630 | | 0.0158 | 0.0139 | 0.3473 |

From Table 1, we can see that the Optimized version has a very significant speedup relatively to the basic OpenCL version, including data transfer and kernel execution no matter which size. The speed of the optimized OpenCL version is improved approximately 200% compared with the CPU version.

In the optimized OpenCL version, the data copy spend little time in memory access and time can be ignored. This result highlight the importance of that GPU and CPU share Last Level Cache (LLC). Due to this reason, data transfer between devices can easily achieve the really zero-copy.

Moreover, the data-width has the fastest kernel execution time. It has improved roughly 60% faster than the basic OpenCL version. This is reason that we make full use of the SIMD optimization. The entire SIMD-width size is fully filled with the data at once, and this reduces the problem of repeated reading of data and cache miss. No matter basic OpenCL version or optimization version, we already use the SIMD instructions, but we can see that the speedup can be greatly improved by make full use of the SIMD-width size.

To further verify the generality of the optimization scheme, we continue to test two multiprocessor PCs. One has an Intel(R) HD Graphics 530 and an Intel core i7-6700 3.40 GHz CPU and another has an Intel(R) HD Graphics 630 and an Intel core i7-7700 3.6 GHz CPU. Other environments are consistent with previous tests. To evaluate the performance on GPUs, we use the 8-bit gray image of 1920 $\times$ 1080, and all versions were run 50 times. Table 2 shows the execution times.

As can be seen from the table, the optimization scheme greatly improves the execution speed of the algorithm comparing with the CPU version and basic OpenCL version. Because of zero-copy, the read and write actions take almost no time in the optimized OpenCL version no matter which platform. Due to the improvement in GPU performance, the PC with an Intel(R) HD Graphics 530 is about 40% faster in the basic OpenCL version and about 40% faster in the optimized OpenCL than the PC with HD4600 about kernel execution. The PC with an Intel(R) HD Graphics 630 has the less execution time in all OpenCL versions. In the optimized version, the kernel execution speed is improved by

53% than the PC with HD4600 and 32% than the PC with HD530. This also shows that our scheme is possessed of stronger applicability and generality.

## 5 Conclusion

The paper presents an optimized scheme about a parallel implementation of demosaicing algorithm using OpenCL. We detailed describe each step about how the original algorithm is implemented, parallelized and optimized. In addition, we introduce how the algorithm executes on the GPU. Specifically, our optimized scheme makes full advantage the modern parallel computing architecture, which increases the parallelism of the process and reduces the computational complexity and the execution time. We implement a basic OpenCL version and further optimized this version. The results show that optimization version has a significant performance improvement about kernel execution, the optimized OpenCL version is up to 6x and the data transmission time is almost zero. And experimental results shows the good applicability of the optimized scheme.

It confirms that the algorithm should be adapted to OpenCL codes accordingly to the hardware execution environments. Indeed, by optimizing the OpenCL code, a 6 speedup yielded by the Optimized OpenCL version comparing with the basic OpenCL version. For some algorithms, it can be well optimized. OpenCL can play a greater role in heterogeneous computing.

**Acknowledgements.** The work is supported by the Science and Technology Key Project of Tianjin under Grant No. 17YFZCGX01180 and Tianjin Key Laboratory of Advanced Networking (TANK).

## References

1. Wang, J., Wu, J., Wu, Z., Jeon, G.: Filter-based bayer pattern CFA demosaicking. Circ. Syst. Sig. Process. **36**(7), 2917–2940 (2017)
2. Chen, R., Jia, H., Wen, X., Xie, X.: Bayer demosaicking using optimised mean curvature over RGB channels. Electr. Lett. **53**(17), 1190–1192 (2017)
3. Lien, C.Y., Yang, F.J., Chen, P.Y.: An efficient edge-based technique for colour filter array demosaicking. IEEE Sens. J. **PP**(99), 1 (2017)
4. Andrade, D.C.D., Trabasso, L.G.: An opencl framework for high performance extraction of image features. J. Parallel Distrib. Comput. **109**, 75–88 (2017)
5. Tan, H., He, X., Wang, Z., Liu, G.: Parallel implementation and optimization of high definition video real-time dehazing. Multimedia Tools Appl. **76**, 1–22 (2016)
6. Wang, D., Yu, G., Zhou, X., Wang, C.: Image demosaicking for Bayer-patterned CFA images using improved linear interpolation. In: Seventh International Conference on Information Science and Technology, pp. 464–469 (2017)
7. McGuire, M.: Efficient, high-quality bayer demosaic filtering on GPUs. J. Graph. GPU Game Tools **13**(4), 1–16 (2008)
8. Malvar, H.S., He, L.W., Cutler, R.: High-quality linear interpolation for demosaicing of Bayer-patterned color images. In: IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, pp. iii–485–8 (2004)

9. Al-Hashimi, B.M.: Energy-efficient run-time mapping and thread partitioning of concurrent openCL applications on CPU-GPU MPSoCs. ACM Trans. Embed. Comput. Syst. **16**(5s), 147 (2017)
10. Dashti, M., Fedorova, A.: Analyzing memory management methods on integrated CPU-GPU systems. ACM SIGPLAN Notices **52**(9), 59–69 (2017)
11. Jang, B., Schaa, D., Mistry, P., Kaeli, D.: Exploiting memory access patterns to improve memory performance in data-parallel architectures. IEEE Trans. Parallel Distrib. Syst. **22**(1), 105–118 (2011)
12. Holewinski, J., Sadayappan, P.: High-performance code generation for stencil computations on GPU architectures. In: ACM International Conference on Supercomputing, pp. 311–320 (2012)
13. Pereira, P.M.M., Domingues, P., Rodrigues, N.M.M., Falcao, G., Faria, S.M.M.D.: Optimizing GPU Code for CPU Execution Using OpenCL and Vectorization: A Case Study on Image Coding. Springer (2016)