Acceleration of Variance of Color Differences-Based Demosaicing Using CUDA

Muhammad Ismail Faruqi*, Fumihiko Ino†, and Kenichi Hagihara‡
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Email: {*i_faruqi**, *ino*†, *hagihara*‡}*@ist.osaka-u.ac.jp*

***Abstract*—Image demosaicing algorithms are used to reconstruct a full color image from the incomplete color samples output (RAW data) of an image sensor overlaid with a Color Filter Array (CFA). Better demosaicing algorithms are superior in terms of acuity, dynamic range, signal to noise ratio, and artifact suppression, which make them suitable for high quality delivery such as theatrical broadcast.**

In this paper, we present our efforts in examining the feasibility of exploiting the Graphics Processing Unit (GPU) as an emerging accelerator to create an on-the-fly implementation of Variance of Color Differences (VCD) demosaicing, a state-of-the-art heuristic demosaicing algorithm developed to eliminate false-color artifacts in texture region of images.

Our efforts in this paper are 1) implementing the algorithm as several kernels to separate the bottleneck portion of the algorithm from the rest and to minimize idle threads and 2) reducing I/O between shared and global memory when performing green channel interpolation by separating the input RAW data into four channels. We then compare the implementation featuring both acceleration methods with a single kernel implementation. Based on experimental results, our proposed acceleration methods achieved per-frame processing time of 343 ms on an nVidia GTX 480, which translates into 2.95 fps. Additionally, our proposed methods were also able to accelerate the kernel time and the effective memory bandwidth by a factor of 2.1x compared with its single kernel counterpart.

***Keywords*—Parallel processing; image demosaicing; CUDA; GPU**

I. Introduction

Image demosaicing [1] is an integral part of color imaging pipeline. It is the first step of the pipeline, in which the luminance data known as RAW data in each photosite is expanded into RGB values. One method to evaluate the quality of demosaicing algorithms is to measure how effective they approximate the remaining color values while not introducing artifact known as moiré. The less moiré introduced in the demosaiced images, the better quality the demosaicing algorithm is.

However, sophisticated demosaicing algorithms tend to be computationally expensive and impractical to be implemented on almost all digital cameras. Therefore, demosaicing algorithms used in many digital cameras are generally favoring execution speed over quality, resulting pictures with moiré. Hence, to enable users accessing higher quality images, higher-end cameras offer a feature to directly record RAW data into storage without going through the entire imaging pipeline. Such data is then processed by some external processors capable executing demosaicing in shorter time.

In another spectrum, users of video cameras equipped with Bayer filter [2] have suffered by the long time and huge space required to acquire high quality files from their camera. To process the files, users have to wait for a long time until the demosaicing algorithm finishes processing the frames, and then saving the demosaiced file into storage. Here lie the challenges this paper is going to solve: to enable users acquire high quality demosaicing result from high resolution video files quickly without having to store large demosaicing result into storage.

In this paper, we propose an acceleration of Variance of Color Differences (VCD) [3]-based demosaicing, a high quality demosaicing algorithm specifically developed to combat moiré in texture region of images, using Compute Unified Device Architecture (CUDA) [4]. The objective of this implementation is to demosaic video RAW files on-the-fly as fast as possible, so that the video editing workflow will be accelerated and the storage requirement to work on demosaicing result can be eliminated. To achieve this, we first introduce the wavefront processing as the base method of the algorithm parallelization. We then propose two implementation methods, which are 1) implementing the algorithm as multiple kernels to separate the bottleneck portion of the algorithm and to minimize idling threads, and 2) reducing input and output transfer between global and shared memory [4] in the green channel interpolation phase by separating the input RAW data into separate channels.

II. Related Work

Several works have been dedicated to implement demosaicing using GPU. For example, McGuire [5] accelerated Malvar-He-Cutler [6] image demosaicing algorithm using OpenGL in real-time speed. Fung et al. [7] show two examples of CUDA-based demosaicing based on bilinear and Lanczos [8] method. However, these algorithms have inferior moiré suppression compared to Chung's algorithm. The first commercial application known to deliver real-time preview and grading for 4K RAW was IRIDAS [10] Speedgrade XR which was launched

in 2009. However, based on its high speed, we suspect it to perform a demosaicing algorithm with low complexity, although the exact algorithm is closed.

Meanwhile, recent high-quality demosaicing algorithms include Mairal et al. [12] algorithm. This algorithm along with VCD has a high CPSNR figure. However, Menon et al. [13] benchmarks recent demosaicing algorithms including Mairal's and Chung's algorithm, and we found Chung's algorithm has the best CIELab [14] figure which closely resembles human eye evaluation result.

III. Variance of Color Differences (VCD) Demosaicing (Chung's Algorithm)

This section explains VCD-based demosaicing. VCD is a demosaicing algorithm developed by Chung and Chan [3] to suppress artifact known as moiré when demosaicing Bayer CFA. It extends Adaptive Color Plane Interpolation (ACPI) algorithm [9] by heuristically determining the interpolation direction of the green channel. In this paper, we describe Chung's algorithm as an algorithm consisting of four steps explained from section III-A to section III-D.

For the rest of the explanation of Chung's algorithm, we will denote the value of a sampling position from the input RAW data at (x, y) coordinate as $P_{x,y}$. Each $P_{x,y}$ is overlaid by a color filter element whose channel type is either red, green, or blue. The output of the algorithm is an image with complete information in each sampling position, where the red, green, and blue channel values at (x, y) will be addressed as $r_{x,y}$, $g_{x,y}$, and $b_{x,y}$ respectively.

The problem of a demosaicing algorithm for Bayer CFA is to interpolate two missing channel values in each $P_{x,y}$. It needs to interpolate $g_{x,y}$ and $b_{x,y}$ at red CFA sampling positions, $r_{x,y}$ and $b_{x,y}$ at green CFA sampling positions, and $r_{x,y}$ and $g_{x,y}$ at blue CFA sampling positions. By completing all of those channels, we construct a full color image from a RAW image with Bayer CFA.

A. Precalculation of Three Green Channel Values at Red and Blue CFA Sampling Positions

As the first step, the algorithm precalculates green channel values for each pixel on blue and red CFA sampling positions in horizontal ($\hat{g}^H_{x,y}$), vertical ($\hat{g}^V_{x,y}$), and diagonal ($\hat{g}^D_{x,y}$) direction. They are obtained as follows:

$$\hat{g}^H_{x,y} = \frac{P_{x-1,y} + P_{x+1,y}}{2} + \frac{2P_{x,y} - P_{x-2,y} - P_{x+2,y}}{4}, \quad (1)$$

$$\hat{g}^V_{x,y} = \frac{P_{x,y-1} + P_{x,y+1}}{2} + \frac{2P_{x,y} - P_{x,y-2} - P_{x,y+2}}{4}, \quad (2)$$

$$\hat{g}^D_{x,y} = \frac{P_{x-1,y} + P_{x+1,y} + P_{x,y-1} + P_{x,y+1}}{4} + \frac{4P_{x,y} - P_{x-2,y} - P_{x+2,y} - P_{x,y-2} - P_{x,y+2}}{8}. \quad (3)$$

In the actual algorithm, those values are computed on-the-fly while interpolating green channel. However, since those values are used frequently in green channel interpolation phase, moving those values calculation into precomputation will reduce its execution steps from $O(w+h)$ steps into $O(1)$ steps.

B. Interpolation of Green Channel Values at Red and Blue CFA Sampling Positions

The next step, which is the contribution of Chung and Chan is to heuristically interpolate green channel at red and blue CFA sampling positions. First, they classify whether a CFA sampling position belongs to a texture or to an edge. This is achieved by computing an edge classifier $e_{x,y}$, which is obtained from summing the difference of the neighboring pixels in horizontal (L^H) and vertical (L^V) direction. The values of $e_{x,y}$, L^H, and L^V are obtained from (4), (5), and (6) respectively.

$$e_{x,y} = max\left(\frac{L^H}{L^V}, \frac{L^V}{L^H}\right), \quad (4)$$

$$L^H = \sum_{-2\le dy\le 2}\left(\sum_{-2\le dx\le 2, x\neq 0} |P_{x+dx,y+dy} - P_{x,y+dy}|\right), \quad (5)$$

$$L^V = \sum_{-2\le dx\le 2}\left(\sum_{-2\le dy\le 2, y\neq 0} |P_{x+dx,y+dy} - P_{x+dx,y}|\right). \quad (6)$$

The CFA sampling position is classified as an edge if $e_{x,y} \ge T$, where T is a predefined threshold by user. Its green channel value $g_{x,y}$ can then be interpolated according to (7).

$$g_{x,y} = \begin{cases} \hat{g}^H & \text{if } L^H < L^V, \\ \hat{g}^V & \text{otherwise.} \end{cases} \quad (7)$$

On the other hand, the CFA sampling position is classified as a texture if $e_{x,y} < T$. Chung and Chan stated that for a texture, three variance of color differences values can additionally supplied to increase the accuracy of the interpolation direction for this CFA sampling position. Those values are variance of color differences in horizontal, vertical, and diagonal directions which are obtained by computing (8)–(10).

$$ {}_H\sigma^2_{x,y} = \frac{1}{9}\sum_{i\in\Psi}((d_{x+i,y}) - \frac{1}{9}\sum_{j\in\Psi}(d_{x+j,y})), \quad (8) $$

$$ {}_V\sigma^2_{x,y} = \frac{1}{9}\sum_{i\in\Psi}((d_{x,y+i}) - \frac{1}{9}\sum_{j\in\Psi}(d_{x,y+j})), \quad (9) $$

$$ {}_D\sigma^2_{x,y} = \frac{1}{2}(\frac{1}{9}\sum_{i\in\Psi}((f_{x+i,y}) - \frac{1}{9}\sum_{j\in\Psi}(f_{x+j,y})) + \frac{1}{9}\sum_{i\in\Psi}((f_{x,y+i}) - \frac{1}{9}\sum_{j\in\Psi}(f_{x,y+j}))), \quad (10) $$

where ${}_H\sigma^2_{x,y}$, ${}_V\sigma^2_{x,y}$, and ${}_D\sigma^2_{x,y}$ denote variance of color differences in horizontal, vertical, and diagonal directions respectively. The range of those summations is defined by $\Psi = \{0, \pm1, \pm2, \pm3, \pm4\}$. Furthermore, $d_{x+i,y}$, $d_{x,y+i}$, $f_{x+i,y}$, and $f_{x,y+i}$ are defined with (11)–(14).

$$ d_{x+i,y} = \begin{cases} P_{x+i,y} - g_{x+i,y}, & \text{if } i = -4, -2, \\ P_{x+i,y} - \hat{g}^H_{x+i,y}, & \text{if } i = 0, 2, 4, \\ \frac{d_{x+i-1,y}+d_{x+i+1,y}}{2}, & \text{if } i = \pm1, \pm3, \end{cases} \quad (11) $$

$$ d_{x,y+i} = \begin{cases} P_{x,y+i} - g_{x,y+i}, & \text{if } i = -4, -2, \\ P_{x,y+i} - \hat{g}^V_{x,y+i}, & \text{if } i = 0, 2, 4, \\ \frac{d_{x,y+i-1}+d_{x,y+i+1}}{2}, & \text{if } i = \pm1, \pm3, \end{cases} \quad (12) $$

$$ f_{x+i,y} = \begin{cases} P_{x+i,y} - g_{x+i,y}, & \text{if } i = -4, -2, \\ P_{x+i,y} - \hat{g}^D_{x+i,y}, & \text{if } i = 0, 2, 4, \\ \frac{f_{x+i-1,y}+f_{x+i+1,y}}{2}, & \text{if } i = \pm1, \pm3, \end{cases} \quad (13) $$

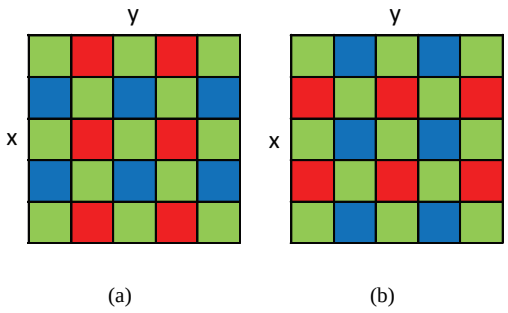
$$ f_{x,y+i} = \begin{cases} P_{x,y+i} - g_{x,y+i}, & \text{if } i = -4, -2, \\ P_{x,y+i} - \hat{g}^D_{x,y+i}, & \text{if } i = 0, 2, 4. \\ \frac{f_{x,y+i-1}+f_{x,y+i+1}}{2}, & \text{if } i = \pm1, \pm3. \end{cases} \quad (14) $$

Finally, the green channel value $g_{x,y}$ can be interpolated by (15).

$$ g_{x,y} = \begin{cases} \hat{g}^H & \text{if } {}_H\sigma^2_{x,y} = \min({}_H\sigma^2_{x,y}, {}_V\sigma^2_{x,y}, {}_D\sigma^2_{x,y}), \\ \hat{g}^V & \text{if } {}_V\sigma^2_{x,y} = \min({}_H\sigma^2_{x,y}, {}_V\sigma^2_{x,y}, {}_D\sigma^2_{x,y}), \\ \hat{g}^D & \text{if } {}_D\sigma^2_{x,y} = \min({}_H\sigma^2_{x,y}, {}_V\sigma^2_{x,y}, {}_D\sigma^2_{x,y}). \end{cases} \quad (15) $$

C. Interpolation of Red and Blue Channel Values on Green CFA Sampling Positions

Using the result of the interpolated green channel, we can compute the values of red and blue channels on green CFA sampling positions. To do this, we interpolate $h_{x,y}$ value horizontally and $v_{x,y}$ value vertically at (x, y) according to (16) and (17).



(a) (b)

Figure 1. (a) The location of green CFA sampling position surrounded by red CFA horizontally and blue CFA sampling positions vertically. (b) The location of green CFA sampling position surrounded by blue CFA horizontally and red CFA sampling positions vertically.

$$ h_{x,y} = g_{x,y} + \frac{P_{x-1,y} - g_{x-1,y} + P_{x+1,y} - g_{x+1,y}}{2}, \quad (16) $$

$$ v_{x,y} = g_{x,y} + \frac{P_{x,y-1} - g_{x,y-1} + P_{x,y+1} - g_{x,y+1}}{2}. \quad (17) $$

Those two values are then assigned as interpolated blue or green channel value, according to the location of the sampling position. If it is located as Fig. 1(a) shows, the missing red and blue values will be assigned as $r_{x,y} = h_{x,y}, b_{x,y} = v_{x,y}$, otherwise if it is located as Fig. 1(b) they will be assigned as $b_{x,y} = h_{x,y}, r_{x,y} = v_{x,y}$.

D. Interpolation of Red Channel Values on Blue CFA Sampling Positions and Blue Channel Values on Red CFA Sampling Positions

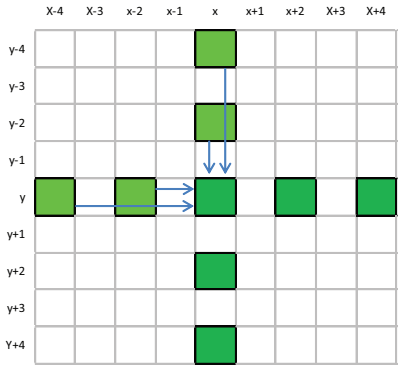
Finally, we interpolate the red channel value at blue CFA sampling positions, and the blue channel value at red CFA sampling positions. The equation for red channel value interpolation is described in (18). The value for the blue channel at red CFA sampling positions is also computed in the same fashion.

$$ r_{x,y} = g_{x,y} + \frac{1}{4}\sum_{dy=\pm1}\sum_{dx=\pm1}(P_{x+dx,y+dy} - g_{x+dx,y+dy}) \quad (18) $$

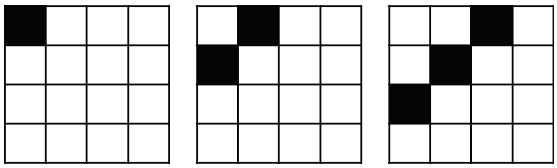
IV. Parallelization Strategies of Chung's Demosaicing Algorithm

In this chapter, we will describe the parallelization strategy we have gone through in accelerating Chung's demosaicing algorithm. We choose CUDA as the platform to accelerate Chung's algorithm since it has the data parallelism, where the GPU has potential to perform it faster than the CPU.

As an overview, we start by explaining the wavefront method which we believe is the natural parallelization method for



(a)



(b)

Figure 2. (a) An illustration of data dependencies when interpolating the green channel at $P_{x,y}$. The light green box indicates interpolated green channel values, and the arrows indicates the direction of the dependency. (b) An example of wavefront processing. Here, one thread is responsible to interpolate the green channel at one sampling position. Threads in the same anti diagonal, i.e. thread wave, are executed in parallel. Meanwhile, each thread waves are executed sequentially.

data dependency introduced by (11)–(14). We then propose two strategies to accelerate Chung's algorithm, which are 1) separating data-dependent portion from the rest of computation by implementing Chung's method as multiple kernels and 2) reducing idling threads during I/O transfer by separating input data into several channels.

A. Parallel Processing by Wavefront Method

In the green channel interpolation phase of Chung's algorithm, (11) and (13) require previous green channel interpolation results of $P_{x-2,y}$ and $P_{x-4,y}$, meanwhile (12) and (14) require previous interpolation result of $P_{x,y-2}$ and $P_{x,y-4}$. This implies a form of data dependency which is illustrated in Fig. 2(a). As a result, $P_{x,y}$ is dependent to the green channel interpolation results at $P_{x-2,y}$, $P_{x-4,y}$, $P_{x,y-2}$, and $P_{x,y-4}$. Therefore, we hypothesize that the natural parallelization method for this phase while guaranteeing the required data dependency is by a wavefront processing method [15] which is illustrated by Fig. 2(b).

To conform with CUDA's hierarchical execution model [4], the implementation of the wavefront method for Chung's algorithm in this paper consists of two levels: the *grid-level* and the *block-level*. Fig. 3 illustrates the grid-level wavefront

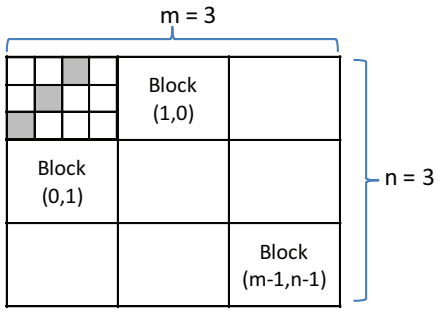


Figure 3. The execution area P is divided into $m \times n$ blocks. Here $m = 3, n = 3, \alpha = 4$, and $\beta = 3$. A block wave D_k consists of all blocks with the same $k = m + n$. For example, block $(0, 1)$ and $(1, 0)$ belong to D_1.

processing. At the grid-level, the RAW data P with width w and height h is divided into $m \times n$ blocks. Each block has width α and height β, hence $m = \lceil \frac{w}{\alpha} \rceil$ and $n = \lceil \frac{h}{\beta} \rceil$. All blocks where $k = m + n$ are grouped into a *block wave* D_k. The application processes the block waves sequentially starting from top left to bottom right, where block wave D_{k-1} is processed before block wave D_k. Each block wave is mapped into a CUDA kernel launch with one dimensional grid size. In CUDA, synchronization between block waves is done automatically between kernel launches.

Inside a block, if a pixel $P_{x,y}$ has $e_{x,y} < T$, it is mapped into a thread. All threads with thread index (t_x, t_y) are grouped as a *thread wave* T_v, where $v = t_x + t_y$. The kernel consists of a loop in which each thread decides to process a pixel or not in the wave depending on its thread index. The thread waves are processed sequentially, where T_{r-1} is executed before T_r. Between each thread wave execution, synchronization is performed once. Fig. 2(b) illustrates the block-level wavefront processing. In CUDA, synchronization between thread waves is accomplished manually by `__synchthreads()` function provided by the CUDA Software Development Kit (SDK).

The computation of thread waves inside a block is performed in $\alpha + \beta - 1$ steps, while the computation of the entire block waves is performed in $m + n - 1$ steps. Hence, this CUDA-based wavefront method takes a total $w + h + m(\beta - 1) + n(\alpha - 1) - (\alpha - \beta - 1)$ steps to interpolate the green channel at blue and red CFA sampling positions. In other words, it has $O(w + h)$ steps which will potentially become the bottleneck of the application.

B. Separation of Data-dependent Portion from the Rest of Computation

Fig. 4 illustrates that although in the green channel interpolation step $P_{x,y}$ depends on the previous interpolation result of its neighbors, the rest of steps are independent and can be accelerated by an embarrassingly parallel method. Hence, if all of four Chung's algorithm's steps are implemented as a single monolithic kernel, the performance of the phases without any data dependency will be hindered by the bottleneck mentioned

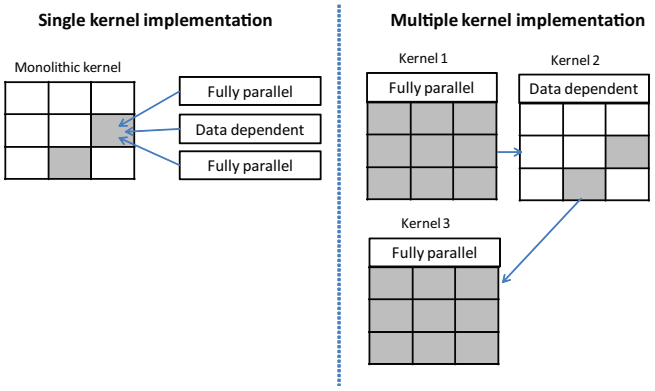


Figure 4. When the algorithm is implemented as a single monolithic kernel, the performance of data-independent portions of calculation will be reduced by the bottleneck portion of the algorithm.

TABLE I
FINAL KERNEL LIST.

Step	Remarks	Area	Kernel
N/A	Transfers data from host to global memory.	R	N/A
—	Converts input from unsigned short to float.	R	1
1	Precomputes temporary green channels.	Q	2
2	Performs edge detection.	P	3
—	Performs the channel separation of input RAW data.	R	4
2	Calculate variance values and interpolate G channel value.	P	5
3 & 4	Interpolate the rest of the channels.	P	6
—	Combine per-channel float-typed output data into unsigned short RGBA data.	R	7
N/A	Transfers data from global to host memory.	R	N/A

before. On the other hand, if the algorithm is implemented as several kernels, the bottleneck portion can be isolated into its own kernel without hindering other steps' performance.

Based on this fact, we propose to classify the computation steps in Chung's algorithm based on their data dependency. The classification policies are:

1) The computation phase with data dependency is separated from the rest and is implemented as one kernel.
2) The computation phases without any data dependency are united together into one kernel.

Based on those policies, because there is only one step with data dependency, we logically need to implement Chung's algorithm as three CUDA kernels: 1) a kernel that performs all computations before green channel interpolation phase, 2) a kernel to perform the green channel interpolation step, and 3) a kernel that handles the rest of computations. However, due to optimization schemes introduced in Sec. IV-C–IV-D, the implementation is finally implemented as seven kernels as shown in Table I.

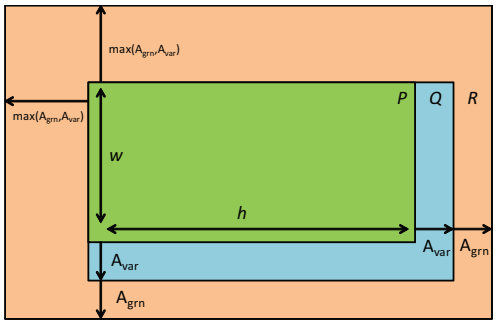


Figure 5. Computation areas. *R* denotes the execution area of input loading phase. *Q* denotes the computation area of precomputation of green channel values phase, where *P* denotes the computation area of other phases.

C. Classification Based on Differences of Kernel Execution Area

Besides the data dependency-based separation policies explained in Section IV-B, another factor that must be considered when implementing a CUDA kernel is that if each kernel handles different computation areas, there will be idling threads which will lead into kernel ineffectiveness. A computation area is defined as an area in which a kernel will store its computation result into.

Fig. 5 illustrates different computation areas of each step. The computation area for step 1 which computes $\hat{g}^H_{x,y}$, $\hat{g}^V_{x,y}$, and $\hat{g}^D_{x,y}$ is denoted by Q. Based on (11) and (13), $d_{x,y}$ and $f_{x,y}$ calculations in step 2 require $\hat{g}^H_{x,y}$ and $\hat{g}^D_{x,y}$ in positive direction with maximum range $A_{var} = x + 4$. Additionally, (12) and (14) implies that step 2 also requires $\hat{g}^V_{x,y}$ and $\hat{g}^D_{x,y}$ in positive direction with maximum range $A_{var} = y + 4$. Hence, Q has width $w + A_{var}$ and height $h + A_{var}$.

Meanwhile, based on (1)–(3), to compute Q, we require values of neighboring pixel within a maximum radius of A_{grn} in horizontal and vertical directions as input. In [3], Chung defines $A_{grn} = 2$. Hence, the required data that must be loaded into global memory [4] and converted to floating point type to accommodate all calculation phases exists in area R, which has width $w + A_{var} + A_{grn} + \max(A_{var}, A_{grn})$ and height $h + A_{var} + A_{grn} + \max(A_{var}, A_{grn})$.

As mentioned above, if different areas are processed by the same kernel, there will be idling threads when computing the phase with least area. Fig. 6 shows an example where threads are idling in block $(0, 0)$ and in block $(m - 1, n - 1)$ when computing P. This problem happens because the kernel with $m \times n$ thread blocks tries to process P, Q, and R in different steps.

To solve this problem, we propose to do further classification of the computation steps based on the following policies:

1) Separate the kernel which contains different computation areas into multiple kernels according to its computation area.
2) Combine any subsequent computation steps which have

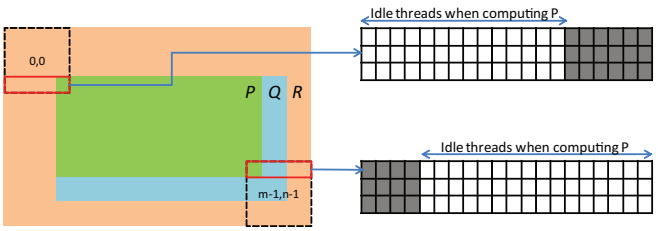


Figure 6. Idle threads that will happen if P, Q, and R are processed by a same kernel. Each cell represents a thread. Gray-colored cells are busy threads, and white-colored cells are idle threads.

the same computation area into one kernel.

Based on the policies above, since kernel 1, 2, and 3 on Table I processes R, Q, and P, we decide to separate those process into three kernels. Similarly, since kernel 6 performs two steps described in Section III-C and III-D in P, we can combine them into one kernel.

D. Reducing Input Loading Time of the Green Channel Interpolation Phase by Separating Input Data

We noticed that in (11)–(14), the only neighbors accessed in input data and output data by $P_{x,y}$ only $P_{x+\alpha,y+\alpha}$, where $\alpha = \pm 2, \pm 4$. Furthermore, all neighboring CFAs of $P_{x+\alpha,y+\alpha}$ have the same CFA as the CFA being computed, and other CFAs are not used.

Based on this fact, in order to save computation time from loading unnecessary CFAs, we propose to separate all values in input data at the same CFA into their own matrix in a separate kernel. Specifically, the input RAW data P is separated into four matrices R, B, GR, and GB in place. Matrix R will hold values from red CFA, matrix GR will hold green CFAs which are surrounded horizontally by two red CFAs, matrix B will hold blue CFAs, and finally matrix GB will hold green CFAs which are surrounded horizontally by two blue CFAs. For P with width w and height h, the width and height of matrices R, B, GR and GB will be $w/2$ and $h/2$, respectively.

Since kernel 5 in Table I that handles green channel interpolation phase is executed multiple times by the wavefront processing, we hypothesize that this optimization will help reduce its total execution time. In this paper, we implement the input separation phase as a kernel which is launched after the edge detection kernel is executed. This kernel is denoted as Kernel 4 in Table I. As the final result of those optimizations, the algorithm is implemented as 7 kernels.

V. Experimental Results and Analysis

In this section, we present the experimental results of our proposed CUDA-based implementation of Chung's algorithm. In order to evaluate our proposed acceleration methods, we set several objectives for our experiments below.

1) Measuring the kernels' performance by comparing execution times among our CUDA-based implementations and a CPU-based of our optimization.

TABLE II
Specification of the experiment machine.

Item	Value
Processor	Intel Xeon X5450, 3 GHz, 4 cores
Memory	DDR3 8 GB
GPU	nVidia GeForce GTX 480
GPU Memory	GDDR5 1.5 GB
Operating System	Windows XP 64-bit XP3
CUDA Version	4.0
IDE	Visual Studio 2005

2) Measuring the efficiency of optimization methods in terms of execution time, floating point operation per second (FLOPS), and memory bandwidth.

The specifications of the machine used for executing the experiments are shown in Table II.

As we do not have access to any video camera capable of outputting a RAW image with 4K resolution, we chose to simulate it by using a RAW image from a still digital camera with nearly similar resolution. The RAW image frame used as input for our experiment has size of 4608 $\times$ 3072 pixels, and uses a Bayer type CFA.

To determine the optimum block width α, width β, and register count r combination for each kernel, we performed the following experiment. First, for each kernel we choose several (α, β) whose static shared memory size [4] fits in a device. Next, using each (α, β) we ran the kernel with an arbitrary number of registers using CUDA Visual Profiler [11] to obtain its static memory size. We then input the obtained static memory size, α, and β into the CUDA Occupancy Calculator [16] to get the biggest register count r_{opt} that results the highest occupancy number. This r_{opt} will give the fastest result for its corresponding α and β. Finally, we compare the results of each (α, β, r_{opt}) combination and chose the fastest one.

A. Performance Evaluation

We used two CUDA implementations, which are a monolithic kernel version and a multiple kernels version which implemented our acceleration methods. The first version performs wavefront processing on all interpolation steps of Chung's algorithm, while the multiple kernels version only performs wavefront processing on the green channel interpolation step, and processes other steps using embarassingly parallel methods. We will refer the monolithic kernel version as the single kernel version hereafter.

We start the evaluation by presenting Table III, which shows the total execution time of our GPU implementations while compared to the CPU implementation which uses a single core. In this table, the total execution time consists of CPU time and GPU time. Furthermore, the GPU time consists of host to device (D $\rightarrow$ H) memory transfer time, kernel time(s), and the device to host (H $\rightarrow$ D) memory transfer time.

Based on Table III, the total execution times of our single

TABLE III
DETAIL OF TOTAL EXECUTION TIME FOR EACH IMPLEMENTATION.

Detail	Execution times (ms)		
	CUDA (multiple kernel)	CUDA (single kernel)	CPU
CPU time	84.5	136.9	5742.0
D → H	13.5	13.5	—
Kernel time	178.9	370.5	—
H → D	65.9	65.9	—
Total time	342.8	586.8	5742.0

kernel and multiple kernel implementations were 587 ms and 343 ms respectively, which are 9.9x and 16.8x faster than its CPU implementation, respectively. As for the CPU implementation, with execution time of 5,742 ms per frame it is definitely not fast enough to demosaic a RAW video stream consisting hundreds of thousands of 4K frames. Even if all of four X5450 cores is used to perform Chung's algorithm in parallel, the execution time would be still slower than our CUDA-based implementations.

B. Efficiency Analysis of CUDA-based Implementations

Two metrics known as memory bandwidth and floating point operations per second (FLOPS) are widely used to measure the effectiveness of CUDA kernels. Given a measured kernel bandwidth b_M, peak GPU bandwidth b_T, measured kernel FLOPS performance f_M, and peak GPU FLOPS f_T, one can calculate kernel's bandwidth effectiveness $E_B = b_M/b_T$ and kernel's FLOPS performance effectiveness $E_F = f_M/f_T$. When $E_B < E_F$ the kernel is said to be *memory-bound*, otherwise the kernel is said to be *arithmetic-bound*. Also, when $E_B \geq 1$, the kernel is said to use the shared memory effectively. The nVidia GTX 480 used for this experiment has a b_T of 177.4 GB/s and a f_T of 1344.96 GFLOPS.

To evaluate the efficiency of the proposed acceleration methods, we chose to analyze and compare the memory bandwidth of both CUDA-based implementations and CPU-based implementations.

Table IV presents the further breakdown of kernel times for the CUDA-based multi kernels implementation. In this table, kernel 5 contains the isolated code of green channel interpolation step. Based on results shown in Table IV, by isolating green channel interpolation in one kernel and eliminating unused input data in that kernel, we are able to reduce the total kernel time of single kernel implementation from 370.5 ms to 178.9 ms, which represents a 2.1x speedup. At the same time, we are also able to increase its memory bandwidth from 7.3 GB/s to 15.2 GB/s which represents a 2.1x increase.

Meanwhile, since the wavefront processing on kernel 5 performs at $O(w + h)$ steps, it only yields E_B of 8.5% even after we applied our optimizations. This kernel actually takes about 85.2% of the total kernel execution time in our multiple kernels version. Therefore, it is still able to benefit from any further memory-based optimizations.

We are also able to let other computation phases to perform

TABLE IV
MEMORY BANDWIDTH COMPARISON BETWEEN CUDA-BASED AND CPU-BASED VCD DEMOSAICING IMPLEMENTATION.

Kernel	Total time (ms)			Bandwidth (GB/s)			E_B (%)	
	Multi	Single	CPU	Multi	Single	CPU	Multi	Single
1	0.9			144.1			81.2	
2	6.2			122.5			69.0	
3	5.5			363.4			200.4	
4	1.6	370.5	5742.0	101.4	7.3	42.6	57.1	4.1
5	152.6			15.2			8.5	
6	9.3			90.3			50.9	
7	2.8			131.8			74.2	

efficiently, which are indicated by their high E_B percentages. For example, the E_B of kernel 3 achieved 200.4%. We observed that such high E_B corresponds with the high memory access count inside the kernel. According to (5) and (6), the total access count for each $P_{x,y}$ is 80. Since those accesses are performed by reading data from the fast shared memory, a high E_B is obtained. This indicates that kernel 3 uses the shared memory effectively.

On the other hand, kernel 2 and 6 only achieved about half of GTX 480 peak memory bandwidth. According to (1)–(3), kernel 2 refers 19 cells for each $P_{x,y}$ to compute $\hat{g}^H_{x,y}$, $\hat{g}^V_{x,y}$, and $\hat{g}^D_{x,y}$, and kernel 6 which minimizes global memory transfer for (16)–(18) also refers 19 cells for each $P_{x,y}$ to interpolate the rest of channels. Theoretically, both kernels should achieve memory bandwidth about a quarter of kernel 3, since their total memory access count is a factor of 0.24 times of kernel 3. Both kernels' E_B confirmed this hypothesis, where the numbers are about a quarter of kernel 3's E_B.

Finally, while the CPU implementation does not seem to be fast enough, it is actually very efficient by looking at its bandwidth number. A memory bandwidth of 42.62 GB/s which far exceeds peak DDR3 bandwidth of 10.6 GB/s by about 400% indicates that 1) the CPU-based implementation uses CPU cache pretty well and 2) performing Chung's algorithm on a 4K frame is very memory intensive.

VI. CONCLUSION AND FURTHER WORK

We have presented a CUDA-based acceleration of Chung's demosaicing algorithm. The acceleration strategy we proposed consists of two methods, which are 1) implementing the algorithm as multiple kernels to separate the bottleneck portion of the algorithm from the rest and to minimize idle threads, and 2) reducing I/O between shared and global memory when performing the green channel interpolation step by separating the input RAW data.

Our experimental results show that methods 1) and 2) succeeded in accelerating the kernel time by a factor of 2.1x faster than its single kernel counterpart. Additionally, we also discover that the green channel interpolation step which becomes the bottleneck of the implementation spends 85.2% of the total kernel time.

As future work, we plan to extend the implementation to

support multiple GPUs usage. We also plan to further increase the performance of green channel interpolation kernel by reducing idle threads during wavefront computation.

ACKNOWLEDGMENT

This work was partly supported by JSPS Grant-in-Aid for Young Researchers (B)(23700057) and Scientific Research (B)(23300007). The authors would like to thank the anonymous reviewers for their helpful comments to improve the quality of the paper.

REFERENCES

[1] B.K. Gunturk, J. Glotzbach, Y. Altunbask, R.W. Schafer, and R.M. Mersereau, "Demosaicing: color filter array interpolation," *IEEE Signal Processing Magazine*, vol. 22, no. 1, pp. 44–54, (2005).
[2] B.E. Bayer, "Color imaging array," U.S. Patent 3 971 065, July 1976.
[3] K.H. Chung, and Y.H. Chan, "Color Demosaicing Using Variance of Color Differences," *IEEE Transactions on Image Processing*, vol. 15, Issue 10, pp. 2944–2955 (2006).
[4] NVIDIA Corporation, "CUDA PROGRAMMING GUIDE 4.0" (2011).
[5] M. McGuire, "Efficient, High-Quality Bayer Demosaic Filtering on GPUs," *Journal of Graphics, GPU, and Game Tools*, vol. 13, issue 4, pp. 1–16 (2008).
[6] H.S. Malvar, L.W. He, and R. Cutler, "High quality linear interpolation for demosaicing of Bayer-patterned color images," *Microsoft Research* (2004).
[7] J. Stam and J. Fung, "Image De-Mosaicing," GPU COMPUTING GEMS EMERALD EDITION, Morgan Kaufmann, pp. 583–398 (2011).
[8] C.E. Duchon, "Lanczos filtering in one and two dimensions," *Journal of Applied Meteorology and Climatology*, vol. 18, pp. 1016–1022 (1979).
[9] J.F. Hamilton and J.E. Adams, "Adaptive color plane interpolation in single sensor color electronic camera", U.S. Patent 5 629 734 (1997).
[10] IRIDAS, http://www.iridas.com (2009).
[11] NVIDIA Corporation: "CUDA Visual Profiler", http://developer.nvidia.com/nvidia-visual-profiler (2011).
[12] J. Mairal, M. Elad, and G. Sapiro, "Sparse Representation for Color Image Restoration," *IEEE Transactions on Image Processing*, vol. 17, issue 1, pp. 53–69 (2011).
[13] D. Menon and G. Calvagno G, "Color Image Demosaicking: An overview," *Signal Processing Image Communication* (2011).
[14] K. McLaren, "The development of the CIE 1976 (L*a*b*) uniform color-space and colour-difference formula," *Journal of the Society of Dyers and Colourists*, vol. 92, pp. 338–341 (1976).
[15] M. Snir, "Resources on Parallel Patterns", http://www.cs.uiuc.edu/homes/snir/PPP/ (2011).
[16] NVIDIA Corporation: "CUDA Occupancy Calculator", http://developer.download.nvidia.com/compute/DevZone/docs/html/C/tools/CUDA_Occupancy_Calculator.xls (2011).