IEICE TRANS. INF. & SYST., VOL.E99–D, NO.12 DECEMBER 2016

PAPER

Performance Optimization of Light-Field Applications on GPU

Yuttakon YUTTAKONKIT[†a)], ***Student Member***, **Shinya TAKAMAEDA-YAMAZAKI**[†], ***Member***, ***and*** **Yasuhiko NAKASHIMA**[†], ***Senior Member***

SUMMARY Light-field image processing has been widely employed in many areas, from mobile devices to manufacturing applications. The fundamental process to extract the usable information requires significant computation with high-resolution raw image data. A graphics processing unit (GPU) is used to exploit the data parallelism as in general image processing applications. However, the sparse memory access pattern of the applications reduced the performance of GPU devices for both systematic and algorithmic reasons. Thus, we propose an optimization technique which redesigns the memory access pattern of the applications to alleviate the memory bottleneck of rendering application and to increase the data reusability for depth extraction application. We evaluated our optimized implementations with the state-of-the-art algorithm implementations on several GPUs where all implementations were optimally configured for each specific device. Our proposed optimization increased the performance of rendering application on GTX-780 GPU by 30% and depth extraction application on GTX-780 and GTX-980 GPUs by 82% and 18%, respectively, compared with the original implementations.

key words: *light-field image processing, GPU optimization, GPU architecture*

1. Introduction

Light-field is a concept of using the direction of light together with its intensity. Using light ray direction, we can extract depth and additional side views of objects, and change the focus plane or extend the depth of field of an image. These features provide many applications in both multiple fields and scales. Industrial fabrication uses a single depth camera, instead of a multiple camera system, to detect product defects [1]. Smartphones can produce a post-processed shallow depth of field on an image, giving the impression of a photograph taken by a digital camera [2]. Recent light-field camera modules shrink the size of a system down to the hand-held scale using a lens array rather than a multiple camera system [3]. A lens array, which is attached to an image sensor, scatters the light or the image data from the camera lens to multiple micro-images in one capture as a raw image. The optical diffraction effect on the multiple micro-images manipulates such additional information as the depth of objects.

The computation between spatial micro-images has been accelerated by a GPU as conventional image processing [4]. The single-instruction multiple-thread (SIMT) mechanism of GPU was used to conceal the latency of memory in intensive computation. However, a sparse memory access pattern does not work well with the multiple memory layers and multi-threading mechanism of SIMT. Therefore, attentive optimization that addresses both computation and memory loads is essential.

Since architecture design of GPUs has been gradually improved, the optimization patterns for specific applications are different among the various architectures [5]. Moreover, the fragmentation specifications of on-the-shelf GPU devices also drastically affect computation and memory manipulation, which hampered the single optimization pattern from working well on every device. Therefore, to fully maximize the performance of each GPU device, the optimal configuration and an optimization technique must be carefully chosen for individual devices.

The following are the main contributions of this research.

- We analyzed the original algorithm of rendering and depth extraction applications, particularly their memory access patterns, and proposed optimized memory access patterns that completely facilitate the SIMT mechanism of GPU devices.
- We indicated the optimal implementation of several GPU devices and showed various results due to differences in architectures as well as fragmentation within the same architecture.

The rest of this paper is organized as follows. We introduce the related work and some basic GPU optimization ideas in Sect. 2. Then we explain the background of light-field applications and their implementations and the capabilities of their optimization in Sect. 3. Next, we introduce the details of our optimization method in Sect. 4. In Sect. 5, we describe the evaluation method and its results and then discuss the benefits to other research. Finally, we conclude this research in Sect. 6.

2. Related Work

Since optimization is the most important part of all application implementation, its fundamental idea is provided by manufacturers of GPU devices [6], [7]. However, in more specific cases, some of the optimization methods have been

Manuscript received February 29, 2016.
Manuscript revised July 2, 2016.
Manuscript publicized August 24, 2016.

[†]The authors are with Computing Architecture Lab, Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
a) E-mail: yuttakon-y@is.naist.jp

DOI: 10.1587/transinf.2016EDP7090

found by optimizing the particular applications. In cone-beam back-projection application optimization [8], synchronizing all the threads at the end of each loop reduced the stray load instructions. A warp speculative method, which reduces warp divergence, was applied in stencil application optimization [9]. This method, which corresponds to the loop collapsing method [10], was also explored as an extensible API [11]. We included these two techniques in both our implementation and our optimal configuration explorations.

Lee et al.[12] summarized the optimization strategies by classifying them with computation-bound or memory-bound applications on the previous generation of GPUs. However, as was previously mentioned about the effect of variant architecture [5], these strategies must be adapted and reevaluated in each GPU architecture. A heuristic automatic-tuning framework [13] reduced the development time of configuration space exploration, but only for sparse matrix multiplication applications.

3. Background

The various lens array-based light-field camera systems [14]–[16] have different optical configurations and manipulation algorithms. In this work, we use the rectangular lens-array light-field input data and algorithms [16].

3.1 Light-Field Image Rendering and Depth Extraction

The lens array attached at the front of the image sensor scatters a conventional image to multiple micro-images. The scattered micro-images are placed in the spatial area, where only a partial area of the micro-image (called a patch) is used for a rendering iterative. The resolution size of the patch indicates the focus plane of the resulting image. A small patch produces back focus, and a large patch produces front focus (Fig. 1(a),(b)). However, to regenerate all of the information from the conventional raw images and to produce a realistic

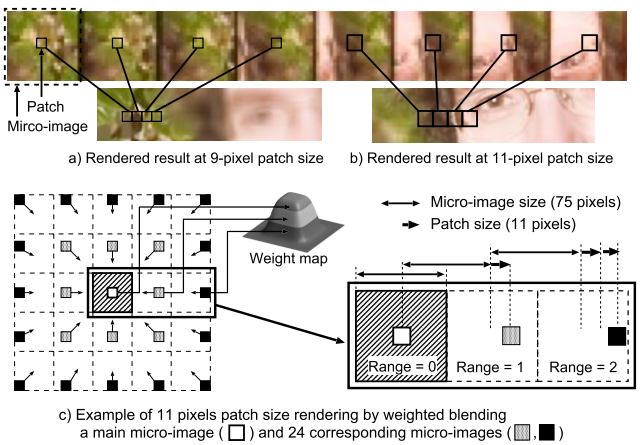


Fig. 1 The resolution of patch size indicates the focus plane of the resulting images. The small patch size produces back focusing (a) and large patch produces front focusing (b). Multiple micro-images are weighted blended together to make a realistic blur effect (c).

Algorithm 1 Light-field rendering kernel

D : micro-image size (pixel), M: patch size (pixel)
$\{nx, ny\} = \{x, y\}/\mathrm{D}$
$\{nxi, nyi\} = \{x, y\}\%\mathrm{D}$
$center = (\mathrm{D}-\mathrm{M})/2$ //shift offset to the center of micro-image
$lim = \lfloor(\lfloor \mathrm{D}/\mathrm{M} \rfloor - 1)/2\rfloor$ //range limit of corresponding micro-images
$\sum pixel = \{0, 0, 0\}$; $\sum weight = 0$;
for {i,j} = $\{-lim, -lim\}$ **to** $\{lim, lim\}$ **do**
//traversing corresponding micro-images
$\{fx, fy\} = \{nx, ny\}*\mathrm{D}+\{nxi, nyi\} + \{center, center\} + \{i, j\}*\mathrm{M}$
$\{wfx, wfy\} = \{nxi, nyi\} + \{center, center\} + \{i, j\}*\mathrm{M}$
$pixel = input[fx, fy]$
$weight = weight\_map[wfx, wfy]$
$\sum pixel \mathrel{+}= pixel * weight$
$\sum weight \mathrel{+}= weight$
end for
$output[x, y] = \sum pixel / \sum weight$

blur effect, the data from the corresponding micro-images have to be weighted blended with the data from the main micro-image (Fig. 1(c)). Each micro-image has its own coordinates relative to the main micro-image on a micro-image plane. The data at the center of the main micro-image will appear in the corresponding micro-images at a position gradually shifting away from the center, the distance of each shift being the patch size times the relative coordinate of the micro-image. As an example with an 11-pixel patch size rendering, the man's eye in Fig. 1(b) shifts gradually away from the center toward the border of the micro-image and then disappears. The number of micro-image steps through which the eye's image shifts until disappearing indicates the range limit of the corresponding micro-images. In this case the range is limited to 2. Calculating in all directions, this rendering extracts the data from a total of 25 micro-images. The weight value of each pixel is defined by its position in relation to its own micro-image, which was pre-calculated by Gaussian distribution as a 75x75 pixels weight map. By shifting the weight value to a position away from the center position of the micro-images, we can change the perspective view of a rendered image, as with a 3D object.

The rendering kernel is designed by an aspect of the render output position per GPU thread (Algorithm 1). The

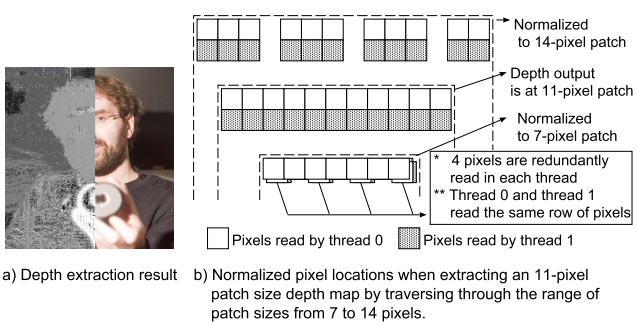


Fig. 2 a) Depth extraction result is illustrated by patch size that renders the best clarity resulting image for each location. b) Multiple iterations of patch sizes are conducted to find the resulting patch size. In each patch size iteration, the target pixel location are normalized in both horizontal and vertical axes to match with the area covered by the patch size.

coordinates of each pixel are translated to the location of the micro-images (*nx*,*ny*) and their relative positions (*nxi*,*nyi*). Then iteration traverses through all the corresponding micro-images ranged by (*-lim*,*-lim*) to (*lim*,*lim*). In each loop iteration, the coordinates of both input and weight data are calculated, loaded and accumulated for averaging.

As shown in Fig. 1(a) and (b), the sharp in-focused object in the rendered image is only produced by blending identical data, and the blurred area are results from dissimilar data. Depth information is extracted using an inverted method of rendering, instead of blending the data from multiple corresponding micro-images. These data are compared to find the least sum-of-absolute difference (SAD) for each location, which indicates the most proper patch size and thus the depth of objects. The grayscale depth map (Fig. 2(a)) is illustrated by the resulting patch size that renders the best clarity in the resulting image for each location. Figure 2(b) shows a memory access example for depth extraction with an 11-pixel output patch size. The horizontal and vertical coordinates of the input pixel from both the main and corresponding micro-images must be normalized to the traversing patch size. For a larger patch size (14 pixels), the pixel coordinates are scattered to cover a whole area of interest and are redundantly read with a smaller patch size (7 pixels) by the same thread or by multiple threads. We extend the depth extraction implementation from the rendering algorithm by adding a patch size traversing iteration at the outermost loop to keep the low register usage of the kernel.

3.2 Overview of GPU SIMT Model and Original Algorithm Implementation

The SIMT converges the context switching of multi-threading and single-instruction multiple-data to obscure the memory latency. Developer defined both the amounts of the application threads and the size of a thread block. The 32 threads in a thread block are each automatically grouped into a warp. All threads in a warp simultaneously execute the same instruction in order to reduce the instruction loading and to coalesce the loading memory address. The coalesced loading reduces the number of transactions and increases the memory bandwidth utilization.

Figure 3 shows the memory hierarchy of the Kepler architecture GPU device. The global memory and the L2 caches are shared among multiple stream multi processors (SMs). The L1 cache and the shared memory share the same memory space and can be configured to match the application requirements. The read-only memory unit, which is also used by texture unit accessing, has its own memory up to 48 KB. The registers act as a gateway for the application to utilize these memory units. Both register and shared memory must be well configured to increase the eligible warps and occupancy of a GPU. By default, the global memory is serviced by a 128-byte cache-line to the L2 cache, and then the L1 cache fetches the data at the same cache-line size. However, the light-field application memory access is too sparse and does not fully utilize this cache-line size. Therefore, the original algorithm utilized the texture unit to access the memory with a 32-byte cache-line but with a slightly more latency. The global memory store transaction is serviced with a 32-byte cache-line directly to the L2 cache.

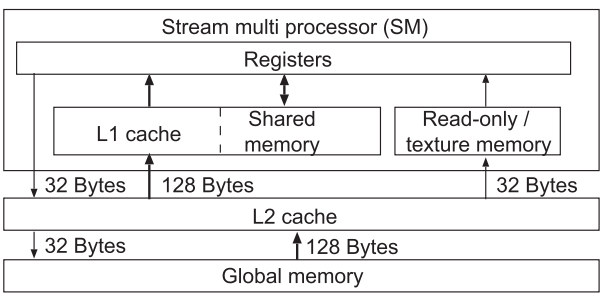


Fig. 3 Memory hierarchy of Kepler architecture GPU device, where numbers indicate cache line size when memory transaction is associated with specific memory units.

3.3 General Optimizations for Original Algorithm

The original implementation can applied these general optimizations as follows.

Vectorizing the input data reduces the cache line miss rate by inserting a blank byte to the data of each pixel (Fig. 4(a)). The loading addresses are always at a quad-byte granularity and have less chance of misalignment than a triple-bytes input pattern, where the loading address can be determined at any byte address. With vectorized data, we can efficiently use the SAD SIMD video instruction (*vabsdiff4*), which compares the SAD value of quad-byte-value and then adds to the destination value. This assembly instruction can be directly inserted into the C++ code by the function of the Parallel Thread Execution ISA [17].

Vectorizing the output data (*ResVect*) reduces the instructions that compute the resulting coordinates. Additionally, we can merge the triple store transactions for each RGB channel into a single quad-byte transaction by adding an additional instruction for the blank fourth value. This can be identified using the NVIDIA profiler or a disassembler.

The GPU application can apply a loop-unrolling tech-

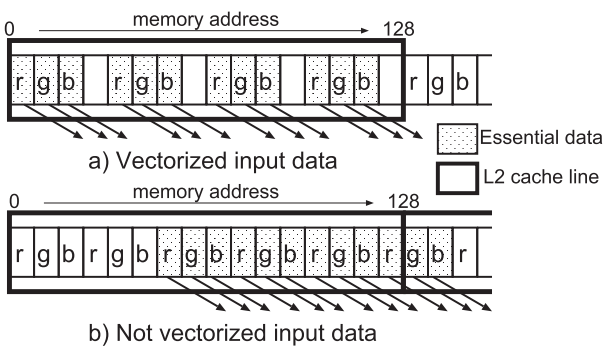


Fig. 4 Vectorized input data (a) have less chance of misaligned loading at L2 caches than the non-vectorized data (b).

nique by trading-off the increased amount of concurrent memory accesses and instruction level parallelism with the cost of additional registers per thread [18]. For both applications, we examined this technique by pre-defining the range limit to 2. Although the compiler detected the constant loop index, it used a predication instead of unrolling the loop. We also applied the macro-unrolling command and found performance degradation due to the register starvation. Thus, we did not apply the loop unrolling in the original implementations.

Since the divide instruction throughput of the GPU is low, we attempted to replace it with the multiply instruction and a set of pre-defined inversed value of dividers (*DivAsMul*). However, this technique costs more memory bandwidth since all of the inversed values of possible dividers (sums of the weight) have to be loaded.

The synchronization at the end of each loop technique (*SyncAtFin*) keeps all the threads in a thread block to load the input data simultaneously, but it also increases the synchronization time.

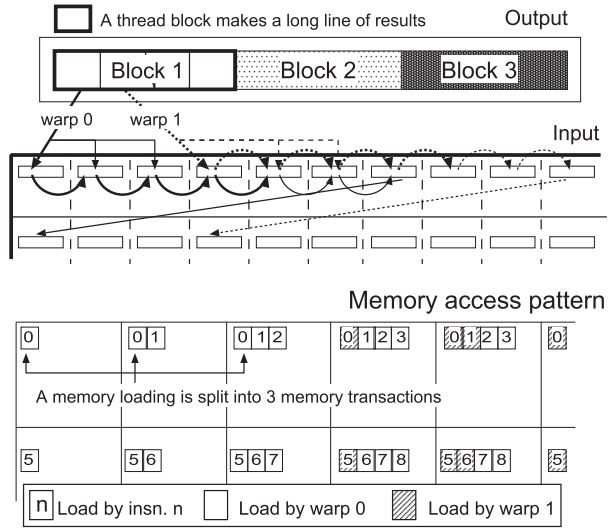
4. Proposed Optimization Implementation

4.1 Memory Access Pattern Optimization

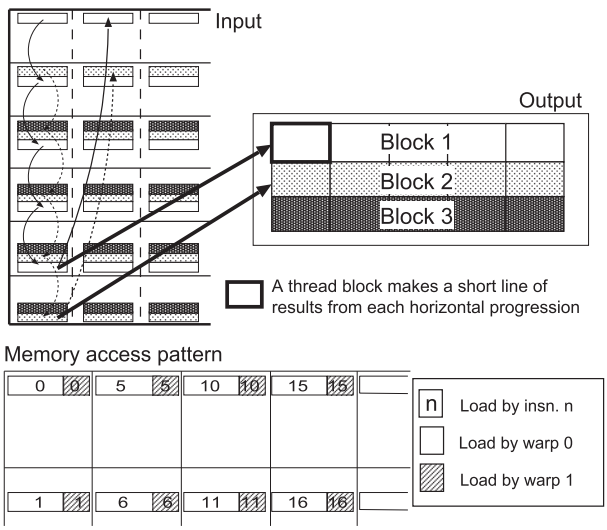
Figure 5 compares the original implementation and the proposed optimization. The Input and Output figures illustrate the implementations' thread blocks arrangement and how they access the input data. The Memory access pattern figures illustrate more detail of the access pattern at raw image data, where a grid-block illustrates for a micro-image, the numbered n blocks illustrate the memory location load by instruction n, and the white-block and pattern-block illustrate the loading from different warp.

Since the original implementation assigned the thread blocks on the basis of the output back to the input, the contiguous pixel computation threads are grouped into the same thread block to achieve a long line of results; and multiple thread blocks are horizontally assigned. A thread block is then divided into multiple warps, as shown in Fig. 5(a), where 128 threads are divided into four warps. In the case of a 11-pixel patch size, a warp contains nearly three of the 11-pixel groups (11+11+10 pixels) that have to load the data from three micro-images per load instruction execution (See arrows labeled warp 0 and warp 1). Thus, each loading is split into three separated memory request (white-blocks numbered 0), which do not efficiently use the cache line at either 128 bytes of the default L2 cache line or 32 bytes of the read-only/texture memory. Although the following load instruction (white-blocks numbered 1,2,3) or load instructions from other warps (numbered patterned-blocks) can use the excess loaded data, the number of memory transactions remains high.

Therefore, we proposed an optimization implementation method by assigning the thread block on the basis of the input to the output (Fig. 5(b)). In the same case of 11-pixel patch size, the corresponding micro-images are at the range



a) Memory access pattern of the original implementation



b) Memory access pattern of the proposed optimization

Fig. 5 Comparison of original implementation (a) and proposed optimization implementation (b).

limit of 2, which produces a total of 11 x 5 = 55 pixels per row. By assigning a pixel computation per thread, a thread block is 55 threads wide. Each thread block is assigned to a group of specific rows of input and a range of horizontal processing micro-image width. The multiple thread block can be assigned to another row of the same micro-image for vertical parallelism or assigned to the same row but to a different micro-image for horizontal parallelism. The white, gray, and dark-gray blocks in the Input figure show the specific rows for thread blocks 1,2,3 in the Output figure.

A thread block loads a line of 55 pixels from each vertical corresponding micro-image (blocks numbered 0,1) and progresses to the next horizontal micro-image (blocks numbered 5,6). Since the corresponding pixels remain scattered on the horizontal corresponding micro-images, each step of

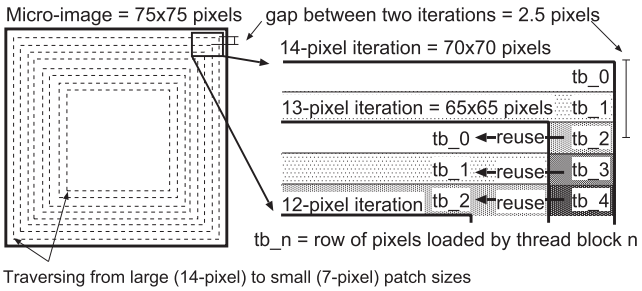


Fig. 6 Data reusability in depth extraction optimization implementations.

the horizontal progression does not gather all of the essential information for all of the threads. Therefore, a thread block manipulates the loaded pixels and stores them in the local memory as a buffer or writes back if all data from the corresponding pixels are accumulated as a short line of results. This method ensures that a line of pixels is always loaded by a coalescing loading.

For depth extraction, the loaded data has both spatial locality from the redundant read when conducting the small patch size iteration (Fig. 2(b)) and the temporal locality among the thread blocks as shown in Fig. 6. Traversing patch size to find the least SAD value generates the multiple iterations. The 14-pixel iteration with corresponding micro-images at a range limit of 2 covers the area of (14x5)x(14x5) pixels, which includes the (13x5)x(13x5) area used by the 13-pixel iteration and give an 86% chance of reusability. Since the gap between two areas used by these iterations is 2.5 pixels, the rows of pixels accessed by thread blocks 2,3,4 at the iteration of the 14-pixel patch size will be accessed and reused by thread blocks 0,1,2 in the next iteration of the 13-pixel patch size. However, the original algorithm traverses the patch size at the outermost loop; therefore, the loaded cache line is expected to have already been replaced before it is reused. In our method, we iterated the traversing from large to small patch size before progressing to the next vertical corresponding micro-images. Thus, the pixels used in the next patch size iteration are expected to have already been loaded by the previous iteration of the other thread block.

4.2 Local Memory Management Scheme

Due to the major role of the local memory in each thread block, we proposed two memory management schemes for our optimization as shown in Fig. 7. The threads are classified into five groups (t0,t1,. . . ,t4) based on their thread ID and patch size.

The thread synchronization scheme (Sync.) (Fig. 7(a)) uses shared memory to store the temporal values. Each thread loads the fixed relative pixel location (0,1,..,4) based on its own group. The loaded data are manipulated and stored in the shared memory as partial results, and then the target shared memory pointers are shifted left for the next horizontal progression (micro-image 2). The fifth thread

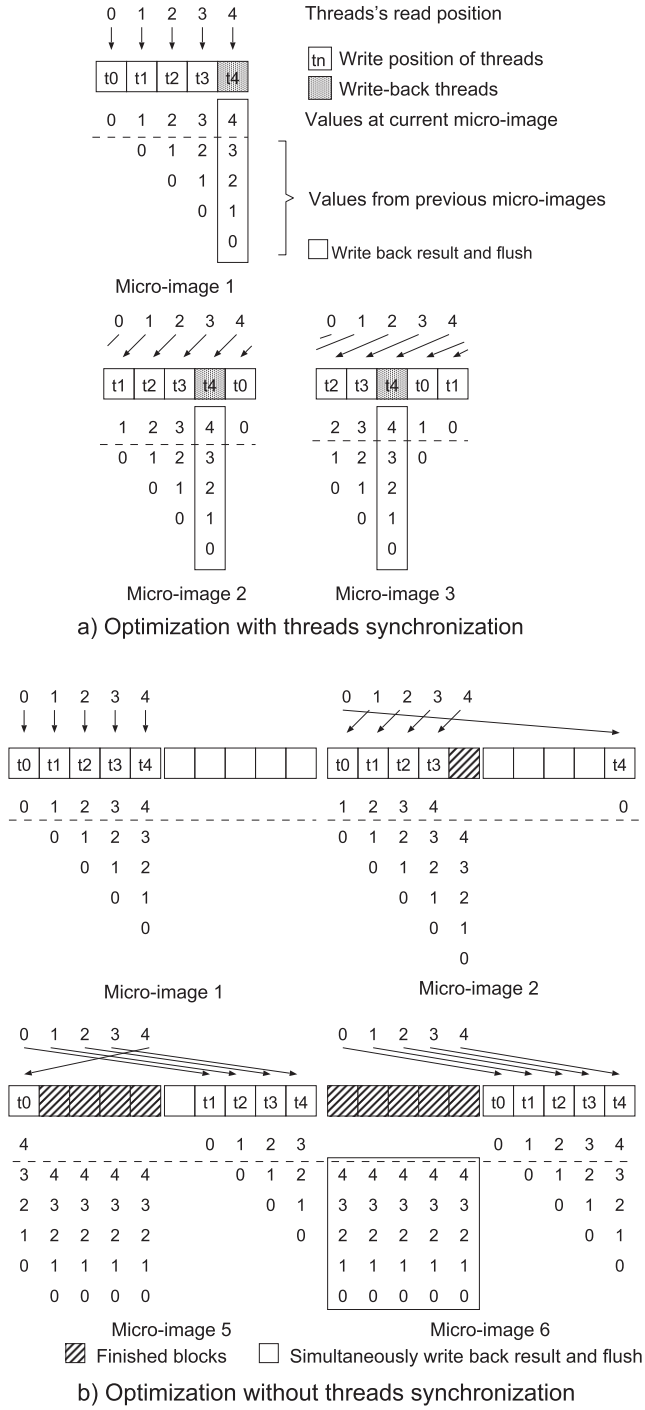


Fig. 7 Proposed local memory management schemes.

group (t4), which loads the pixel from the right-most pixel location, writes back the result and flushes the target shared memory (square block). While t4 is writing the result, the other thread groups are held by the synchronization to avoid a race condition in the shared memory. However, this specified write-back thread group (t4) makes a warp divergence that forces the GPU to serialize the instruction stream, and the synchronization point delays the other threads.

To reduce these problems, we proposed another opti-

mization scheme without thread synchronization (NoSync.) (Fig. 7(b)). This scheme allows all the threads to write back the results at the same time without warp divergence or synchronization. In contrast with Sync., NoSync. threads do not shift the target local memory; instead they shift the relative loading pixel locations. Thus, each thread has its own privileged local memory, which can be implemented by either register or shared memory. To simultaneously conduct writing back from all the threads, we doubled the local memory in order to buffer the finished value. At micro-image 1, t4 loads the data from location 4 that fulfill the block, and then it changes the target local memory to another location but still retains its privileged. At micro-image 2, all groups shift the pixel loading location, where t4 loads data from location 0 and stores them in the empty local memory. When micro-image 6 is reached, t1 - t4 have already changed the target local memory, and only t0 still stores the result on the left side of the local memory. Finally, at Micro-image 7, all the groups have changed the target local memory. They simultaneously store the values and write back the buffered values that have already been completed.

The NoSync. implementation has high register pressure due to the doubled local memory size and more program control flow registers. Therefore, we implemented the local memory by shared memory instead of registers to alleviate this pressure. We evaluated both the default and read-only/texture memory accesses, and both methods are allocated with pitch value for the best access latency in the row order. The memory pointers in the default memory access are decorated with the *__restrict__* keyword to let the compiler reduce the sub-instructions without concern for the data dependency and also to let it enable the direct access to read-only memory for capable devices. However, this optimization costs more registers per thread, which affects the device occupancy. To increase the parallelism, we increased the horizontal dimension of thread blocks by shortening the range of the micro-images of each thread. However, each non-zero thread block has to collect the corresponding data from the four corresponding micro-images to the left before it reads the data from its initial micro-image. These different execution paths among the thread-blocks are written separately on the source file to reduce the thread divergence and the unessential conditional instruction. We found that synchronizing all the threads after the initialization phase also significantly improved the performance.

5. Evaluation

To evaluate our proposed optimization, we configured both the original and our proposed implementations at the most optimal configurations. For the original algorithm implementation, we made a configuration space exploration to find the most optimal configuration for each GPU device. For the proposed optimization implementation, we manually tuned the performance, especially at the register, at the shared memory and at the loop unrolling.

The compiler and environmental setup are provided in Table 1. We used the cudaEvent library to measure the execution times by acquiring the times before and after the kernel call. The data transaction time from the host to the device and vice-versa was not included. We also invoked a dummy kernel to mask an initialization delay of the reference API before starting the cudaEvent, and then we repeated the kernel and averaged the execution times.

We used two sample applications included in CUDA SDK [19] to measure the reference values of memory bandwidth on each GPU device. The *bandwidthTest* application measures the peak value (100%), and the *transpose* application provides a read-then-write only kernel value (Simple copy) and the value of the most optimized matrix transpose kernel (Matrix transpose).

5.1 Different Setup Configuration of Several GPU Devices

We evaluated the performance improvement on the GPU devices from both Maxwell [20] and Kepler architectures as described in Table 2. GTX-650Ti-boost and GTX-670 can use the registers up to 63 registers per thread. Tegra-K1 is a mobile system-on-chip, which includes a GPU and a CPU. The GPU shares the same memory space with CPU and has fewer texture addressing units.

GTX-980 is a Maxwell architecture GPU that provides dedicated 96-KB shared memory, simpler warp schedulers and increased number of active blocks per SM. However, the unified memory for the L1 cache and read-only/texture memory is halved to 24 KB. These adjustments were designed to increase the computation throughput that had not been balanced with the high memory bandwidth in the Kepler architecture.

The configurations for the original implementation are described in Table 4. The thread-block width of Kepler GPUs are optimal at not more than 128 threads with 32 registers per thread, but Maxwell GPU (GTX-980) could improve the performance up to 320 threads with rendering. The vectorized results (*ResVect*) are not significant for such relatively high ratio of computation per memory throughput devices as GTX-980 and Tegra-K1. Replacing the division by multiplication (*DivAsMul*) improved the rendering performance in all the desktop Kepler GPU devices. The synchronization at the end of the loop technique (*SyncAtEnd*) can be applied, but, due to the short life-span of a thread, this did not significantly impact the performance.

The configurations of the proposed implementations for rendering and depth extraction applications are described in Table 5 and Table 6, respectively. The block width and grid size correspond to the patch size and the output resolution of the applications (Table 3). The range of the micro-images indicates the number of micro-images that are horizontally traversed by each thread. Texture unit is used in some rendering implementations to replicate the direct access to read-only memory and in some depth extraction implementations to alleviate the separated access of the large patch size iteration (14 pixels), which is larger than the output patch size (11 pixels). The configurations be-

Table 1 Compiler and environmental setup

Parameters	Description
CUDA library version	CUDA Toolkit 7.5 and 6.5 (Tegra-K1)
Compilation option	-arch=sm_{30,32,35,52} -rdc true
Option for n register limitation	-Xptxas -v –maxrregcount n
Host operation system	Centos 6 64-bit, Linux For Tegra R21.4
GCC library version	GCC 4.4.7 and 4.8.4 (Tegra-K1)

Table 2 Specification of evaluated GPU devices

Devices	**GTX 670**	**GTX 650**[1]	**Tegra K1**	**GTX 780**	**GTX 980**
Stream multiprocessor (SM)	7	4	1	12	16
Core counts	1344	768	192	2304	2048
Core frequency (MHz)	1058	1098	852	902	1253
L2 cache size (KB)	512	384	128	1536	2048
Memory bus (bit)	256	192	64	384	256
Memory frequency (MHz)	3004	3004	924	3004	3505
Compute compatibility	3.0		3.2	3.5	5.2
32-bit registers per thread	63		255		
Read-only memory per SM (KB)	48				24
Shared memory per SM (KB)	64 (shared with L1)				96
Warp schedulers : cores ratios	4:192				1:32
Active block per SM	16				32

[1] GTX-650Ti-boost

Table 3 Parameters of light-field application

Parameters	Description
Micro-image (MI.) size (pixel)	74.75
Patch size (pixel)	11
Range corresponding mirco-image limit	2
Patch size traversing range for depth extraction	8 (7 - 14)
Micro-image count (width x height)	96 x 72
Output resolution (pixel x pixel)	1056 x 792

Table 4 Configuration of original implementation

Application	Rendering					Depth extractaion				
GPU device[2]	M9	G8	G7	G6	TK	M9	G8	G7	G6	TK
Block width	320	128	128	128	160	64	128	96	96	96
ResVect	√	√	√			√	√	√		
SyncAtEnd	√		√		√	√	√	√		√
DivasMul	√	√	√			n/a				

Table 5 Configuration of proposed rendering implementation

	Sync.					NoSync.				
GPU device[2]	M9	G8	G7	G6	TK	M9	G8	G7	G6	TK
Block size	55x1	55x2				55x1				
Grid size	1x792	2x396				2x792				
Range MI.	96	50				50				
Register limit	x	48	40			x	60	48		
Texture unit	x	√				x				
ResVect	x	√				x	√			
L1/Shared	-/96	16/48				-/96	16/48			
Unrolling	low					max				

Table 6 Configuration of proposed depth extraction implementation

	Sync.					NoSync.				
GPU device[2]	M9	G8	G7	G6	TK	M9	G8	G7	G6	TK
Block size	55x1					55x1				
Grid size	2x792					2x792				
Range MI.	50									
Register limit	48	64	63			80	63		64	
Texture unit	x	√				x	√			
ResVect	x	√				√				
L1/Shared	-/96	48/16				-/96	16/48			
Unrolling	low					low	specific		low	
SAD type	int	short				int	short			

[2] M9=GTX-980, G8=GTX-780, G7=GTX-670, G6=GTX-650Ti-boost, TK=Tegra-K1

tween the L1 cache and the shared memory of GTX-980 are unnecessary, given to the fixed-size dedicated shared memory. Kepler GPUs are configured differently because the L1 cache space is also used for the register spilling that frequently occurs when we limit the register per thread to increase the occupancy. The vectorized result is configured for better results, but it did not show a clear relation to the GPU specifications. The register limit configurations were mostly conducted on the Kepler GPUs, not the Maxwell GPU, due to the high cost of register spilling that was previously found in [21]. We also shortened the SAD variable type to 16 bits to reduce the shared memory pressure in Kepler GPUs. The unroll value indicates the loop unrolling, where we found that Sync. scheme is optimal only with the most inner loop unrolled, due to the increasing cost of the synchronization overhead. Since the synchronization occurs less in NoSync. scheme, it is capable of high unrolling, which increases the concurrent memory loading and omits the costly warp divergence caused by the loop control flow instructions. Specific unrolling of the pre-initialization phase loop significantly improved the performance on some Kepler GPUs. We also attempted to reuse the accumulated weight data and replace the division instruction in rendering. However, the performance was reduced due to the increased bank conflict when flushing the shared memory.

We investigated the problem of shared memory bank conflict, especially for the rendering optimizations where the RGB data (and the accumulated weight value as the fourth channel) could be declared as four structures of an array (SoA: int [4][55]) or an array of a structure (AoS: int4 [55]). Since the weight value multiplies each color channel separately, the SoA version did achieve lower bank conflict and less average transactions per request. However, the AoS version achieved better performance, due to the compiler optimizing the four 32-bit load/store instructions to a single 128-bit load/store instruction (LDS.128 and STS.128), which reduces the total shared memory request. Additionally, since all 32 threads in a warp access the shared memory by the 128-bit memory load instruction, all 32 banks of shared memory are fully utilized by the load instruction from 8 threads per access.

5.2 Result and Discussion

Table 7 and Table 8 show the statistic information of rendering and depth extraction implementations, respectively. Our optimization conceals the memory's latency by keeping the memory manipulation near to the computation unit as is shown by the reduced memory dependency stall and increased execution dependency stall. The global store efficiency is also increased due to the improved input data flow. However, more complex implementations require more registers, reducing thus the occupancy and the active threads, and further reduce the IPC, as discussed by [22]. We do not include the global memory load efficiency and the executed and issued load/store instructions information, because these metrics do not measure the memory transactions

Table 7 Statistic information of Rendering application optimization.

Devices	GTX-980			GTX-780			GTX-670			GTX-650Ti-boost			Tegra		
Implementation	Base	Sync	NoSy	Base	Sync	NoSy	Base	Sync	NoSy	Base	Sync	NoSy	Base	Sync	NoSy
Execution dependency stall	19%	17%	12%	20%	23%	27%	-	-	-	-	-	-	26%	15%	21%
Memory dependency stall	70%	51%	71%	46%	16%	9%	-	-	-	-	-	-	28%	17%	1%
Synchronization stall	0%	15%	1%	0%	17%	3%	-	-	-	-	-	-	0%	23%	4%
Achieved occupancy	89%	51%	52%	96%	70%	46%	95%	72%	48%	95%	73%	48%	60%	74%	32%
IPC	2.38	2.31	2.03	3.33	2.86	2.65	3.10	2.73	2.39	3.35	2.80	2.63	3.52	2.87	2.05
Global store efficiency	33%	18%	25%	38%	61%	80%	33%	61%	25%	38%	61%	80%	33%	15%	25%
↘ throughput (GB/s)	11.7	20.9	14.9	8.67	7.84	7.05	5.77	4.70	7.96	1.00	1.31	1.82	0.09	0.41	0.43
↘ transactions (10^3)	235	442	318	52.3	100	56.2	118	100	149	52.3	100	56.2	118	399	79.2
DRAM read (GB/s)	142	138	139	119	141	161	71.1	82.0	78.3	47.3	49.1	49.5	-	-	-
↘ transactions (10^6)	2.89	2.92	2.98	2.88	3.07	3.00	2.90	2.98	3.12	2.89	2.99	3.09	-	-	-
L2 cache hit rate	28%	7%	10%	38%	15%	17%[1]	35%	10%	14%	34%	9%	15%	-	-	-
↘ throughput (GB/s)	197	147	156	193	166	194	109	90.8	90.8	71.6	54.2	58.1	1.65	1.98	-
↘ transactions (10^6)	3.99	3.11	3.32	4.66	3.62	3.62	4.42	3.30	3.62	4.37	3.30	3.62	4.22	3.32	-
Texture cache hit rate	73%	51%	52%	70%	54%	53%	70%	55%	-	70%	55%	-	72%	58%	-
↘ throughput (GB/s)	518	145	148	442	259	295	257	155	-	171	92.8	-	4.08	3.36	-
↘ transactions (10^6)	10.4	3.07	3.15	10.7	5.65	5.49	10.5	5.65	-	10.5	5.65	-	10.5	5.65	-

Table 8 Statistic information of Depth extraction application optimization.

Devices	GTX-980			GTX-780			GTX-670			GTX-650Ti-boost			Tegra		
Implementation	Base	Sync	NoSy	Base	Sync	NoSy	Base	Sync	NoSy	Base	Sync	NoSy	Base	Sync	NoSy
Execution dependency stall	33%	18%	28%	23%	20%	22%	-	-	-	-	-	-	-	-	-
Memory dependency stall	53%	65%	14%	35%	11%	20%	-	-	-	-	-	-	-	-	-
Synchronization stall	0%	3%	1%	1%	7%	1%	-	-	-	-	-	-	-	-	-
Achieved occupancy	74%	60%	36%	74%	49%	33%	75%	49%	42%	74%	50%	42%	73%	50%	43%
IPC	3.15	2.18	3.14	2.79	2.08	1.40	2.61	1.73	1.02	2.68	1.69	1.03	2.06	2.64	1.98
Global store efficiency	33%	61%	80%	38%	61%	80%	38%	61%	80%	38%	61%	80%	33%	61%	80%
↘ throughput (GB/s)	1.33	1.00	0.90	0.67	1.05	0.60	0.44	0.65	0.31	0.26	0.37	0.18	0.01	0.01	0.004
↘ transactions (10^6)	235	171	131	52.3	100	56.2	52.3	100	56.2	52.3	100	56.2	118	100	56.2
DRAM read (GB/s)	75.8	77.3	68.8	31.6	62.4[2]	74.9	35.9	36.8	44.5	20.0	28.2	27.1	-	-	-
↘ transactions (10^6)	13.4	13.6	10.4	10.0	10.2	16.9	17.5	9.86[2]	19.5	16.4	13.3	20.2	-	-	-
L2 cache hit rate	79%	61%	64%	83%	69%	51%	72%	62%	47%	73%	48%	41%	-	-	-
↘ throughput (GB/s)	366	199	187	184	165	131	128	95.0	83.6	75.0	53.6	46.9	2.93	1.32	1.01
↘ transactions (10^6)	64.6	34.8	28.1	58.6	27.0	29.6	62.2	25.5	36.7	61.4	25.2	35.0	64.0	-	-
Texture cache hit rate	49%	56%	57%	54%	62%	59%	51%	63%	58%	51%	63%	60%	50%	-	-
↘ throughput (GB/s)	455	220	192	253	315	224	165	89	36	98	111	81.0	3.65	2.64	2.01
↘ transactions (10^6)	80.3	38.6	28.9	80.3	51.5	50.5	80.3	50.8	60.0	80.3	52.3	60.4	80.3	-	-

- : The information is not available or correctly reported by NVIDIA Profiler.
[1] The higher throughput although with the lower L2 cache hit rate, mentioned in Sect. 5.2, paragraph 1.
[2] The increased memory bandwidth or decreased number of memory transactions, mentioned in Sect. 5.2, paragraph 5.

via read-only/texture memory in Kepler GPUs [23]. Some metrics are not available on some devices or are not reported correctly, especially by Tegra-K1.

Figure 8 shows the effective DRAM bandwidth of our implementations compared to the peak memory bandwidth of each GPU device. The increased computation throughput architecture on GTX-980 achieves better memory latency concealment. This is shown by fully utilizing memory bandwidth in Simple copy and comparing this to other Kepler GPUs, which struggle at less than 75%. For rendering, we achieved, in general, a memory bandwidth equal to or higher than the original implementation. Especially, on GTX-780, with NoSync scheme, we increased the bandwidth beyond Matrix transpose and to the level of Simple copy. The depth extraction application is much more computation-intensive than the rendering and therefore requires a high computation throughput to obscure the memory latency and increase memory bandwidth utilization, as shown by the result from GTX-980.

Figure 9 shows the relative number of memory transactions of our optimizations compared to the original implementations. Our proposed coalesced memory access pattern reduced the number of transactions at read-only/texture memory and, in consequence, at the L2 cache. The L2 cache hit rates are lower because a cache line is expected to be used only once after being loaded. This is also applies to the read-only/texture memory hit rates on rendering implementations. However, the lower L2 cache hit rates do not always imply a lower throughput, if the DRAM throughput is significantly increased, as it is in the rendering optimization of GTX-780 with NoSync. scheme (see Table 7). The data reusability in depth extraction optimizations increased read-only/texture memory hit rates and reduced the number of L2 cache transactions, for example, in the GTX-980 with NoSync. scheme and GTX-780 with Sync. scheme. The performance improvement of our optimizations is the product of increased DRAM bandwidth and decreased number of DRAM transactions.

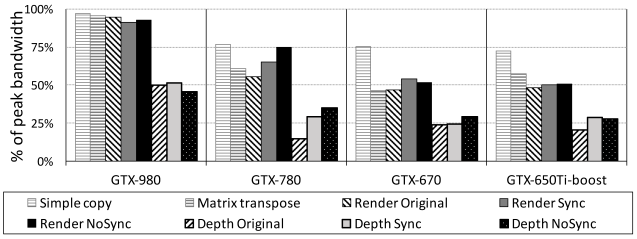


Fig. 8 Effective DRAM read bandwidth of reference applications and all light-field application implementations, compared to the peak memory read bandwidth of GPU devices.

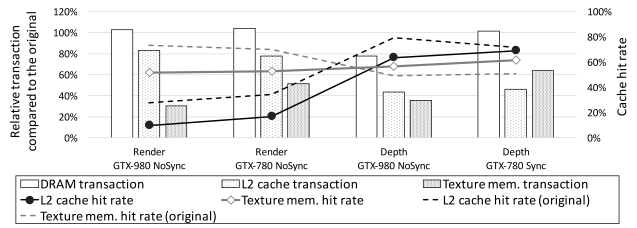


Fig. 9 Relative number of read transactions of DRAM, L2 cache and read-only/texture memory of the optimization implementations compared to the original implementations (bar plots) and their absolute cache hit rates (line plots).

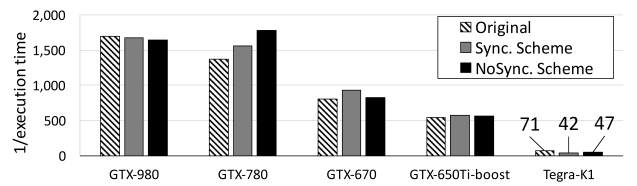


Fig. 10 Optimization result of rendering application.

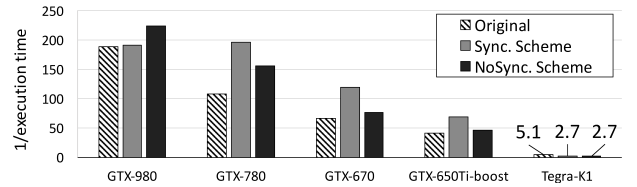


Fig. 11 Optimization result of depth extraction application.

Figure 10 shows the performance improvement of the rendering application. For GTX-980, although we succeeded in reducing the numbers of transactions at both read-only/texture memory and L2 cache, the performance was still throttled by the DRAM memory bandwidth. This indicates that a latency-hiding by parallelism in the original implementation performed better than our latency-hiding by caching (local memory buffering) for a high computation per memory throughput unit such as GTX-980, as discussed by [22], [24]. With the increased DRAM bandwidth, GTX-780 with NoSync. scheme increased the performance by 30% and outperformed GTX-980. However, GTX-670 and GTX-650Ti-boost performed better with Sync. scheme due to the limitation of the direct access to the read-only memory, which is cannot be fully replicated by the texture unit function. Although, the memory can be accessed via the texture unit [7], we found a performance difference between both memory unit function on both GTX-780 and GTX-980.

Figure 11 shows the performance improvement of the depth extraction application. Relocating the patch size traversing loop to the innermost iteration to enable reusing significantly increased the performance, as shown by the 82% performance improvement on GTX-780 and 18% improvement on GTX-980. The NoSync. scheme worked well with GTX-980 but not with Kepler GPUs, where it got the higher numbers of L2 cache transactions than Sync. scheme, which further increased the number of DRAM transactions. Thus, Kepler GPUs preferred Sync. scheme to improve the performance of both read-only/texture memory and L2 cache and to either increase the DRAM bandwidth or decrease the number of DRAM transactions.

The Tegra K1, unlike the other Kepler GPUs, did not increase the performance of rendering application by using Sync. scheme. This was because Tegra K1 had a significant high synchronization stall problem that reduced the performance to below that of the original implementation. For depth extraction optimization, with limited reported information, we could only conclude that the performance degradation was caused by the lower occupancy and drastically decreased L2 cache throughput. Moreover, our proposed method has a thread-block size that is not a multiple of the warp size. This caused ineffective usage of a single SM and reduced the overall performance.

According to our results, determining the most optimal configuration and optimization technique for particular applications and GPU devices requires an empirical evaluation, especially to find the balance among loop unrolling, register control, and L1 cache/shared memory configuration. We encourage other developers to consider unusual methods for example, implicitly replacing the less frequent access register variable with shared memory to reduce the register pressure and increases both occupancy and concurrent memory access. Also, implicitly using the 32-bit floating variable, instead of the integer variable, can reduce the computation latency on some GPUs.

The coordination geometry of the micro-images and their intra pixels can be represented as a 4-dimensional (4D) data. Thus, our proposed optimization method can be applied in the other applications that utilize the 4D data, such as Lattice Quantum Chromodynamics [25].

6. Conclusion

Light-field image processing is a fundamental computation for such approaching applications as re-focusable cameras or the depth of object-based applications. GPU devices are accelerating such computations. However, due to the sparse memory access pattern of the applications, the straightforward implementation of the original state-of-the-art algorithm did not fully utilize the GPU device. Therefore, we analyzed the details of memory access patterns and proposed an optimization method that improves both the computation and the memory access patterns. The proposed optimization fully exploited the benefits of the SIMT mechanism in GPU

devices, effectively utilized all of the available resources, and increased the data reusability in the depth extraction application. We proposed two local memory management schemes, a synchronized scheme (Sync.) which uses less register, and a non-synchronized (NoSync.) scheme which uses more register. To evaluate our proposed method, we first explored the most optimal optimization and configuration for both the original and the optimized implementation based on the several evaluated GPU devices. Compared with the original implementations, our NoSync. scheme improved the rendering performance of GTX-780 by 30% and the depth extraction of GTX-980 by 18%. Sync. scheme improved the depth extraction performance of GTX-780 by 82%.

References

[1] S. Åtolc, D. Soukup, B. Holländer, and R. Huber-Mörk, "Depth and all-in-focus imaging by a multi-line-scan light-field camera," J. Electronic Imaging, vol.23, no.5, p.053020, 2014.
[2] C. Hernández, "Lens blur in the new google camera app." Accessed: 2015-08-6.
[3] R. Ng, M. Levoy, M. Brédif, G. Duval, M. Horowitz, and P. Hanrahan, "Light field photography with a hand-held plenoptic camera," Computer Science Technical Report CSTR, vol.2, no.11, 2005.
[4] A. Lumsdaine, G. Chunev, and T. Georgiev, "Plenoptic rendering with interactive performance using gpus," IS&T/SPIE Electronic Imaging, pp.829513–829513, International Society for Optics and Photonics, 2012.
[5] J. Stratton, N. Anssari, C. Rodrigues, I.J. Sung, N. Obeid, L. Chang, G. Liu, and W. Hwu, "Optimization and architecture effects on gpu computing workload performance," Innovative Parallel Computing (InPar), 2012, pp.1–10, May 2012.
[6] P. Micikevicius, "GPU performance analysis and optimization," 2012.
[7] NVIDIA, "Cuda C programming guide." Accessed: 2015-11-30.
[8] T. Zinsser and B. Keck, "Systematic performance optimization of cone-beam back-projection on the kepler architecture," Proc. 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine, pp.225–228, 2013.
[9] N. Maruyama and T. Aoki, "Optimizing stencil computations for nvidia kepler gpus," Proc. 1st International Workshop on High-Performance Stencil Computations, pp.89–95, Vienna, 2014.
[10] S. Lee, S.J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," SIGPLAN Not., vol.44, no.4, pp.101–110, Feb. 2009.
[11] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," Proc. 2011 international conference for high performance computing, networking, storage and analysis, p.12, ACM, 2011.
[12] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, and A.W. Toga, "Cuda optimization strategies for compute-and memory-bound neuroimaging algorithms," Computer methods and programs in biomedicine, vol.106, no.3, pp.175–187, 2012.
[13] W. Abu-Sufah and A.A. Karim, "Auto-tuning of sparse matrix-vector multiplication on graphics processors," Supercomputing, pp.151–164, Springer, 2013.
[14] C. Perwass, "The next generation of photography," White Paper, www.raytrix.de.
[15] Lytro, "Lytro Illum specification." Accessed: 2015-12-05.
[16] T. Georgiev and A. Lumsdaine, "Focused plenoptic camera and rendering," J. Electronic Imaging, vol.19, no.2, p.021106, 2010.
[17] NVIDIA, "Parallel thread exuction ISA application guide." Accessed: 2015-11-30.
[18] G.S. Murthy, M. Ravishankar, M.M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp.1–11, IEEE, 2010.
[19] NVIDIA, "Cuda sample program." Accessed: 2015-11-30.
[20] NVIDIA, "Maxwell tuning guide," 2012. Accessed: 2015-12-10.
[21] P. Bialas and A. Strzelecki, "Benchmarking the cost of thread divergence in cuda," arXiv preprint arXiv:1504.01650, 2015.
[22] U.B. Vasily Volkov, "Better performance at lower occupancy." Accessed: 2016-6-8.
[23] NVIDIA, "Nvidia gameworks, nvidia nsight visual studio edition, pipe utilization." Accessed: 2016-6-8.
[24] N. TONY SCUDIERO, "GPU memory bootcamp II: Beyond best practices." Accessed: 2016-6-8.
[25] F. Winter, M. Clark, R. Edwards, and B. Joo, "A framework for lattice QCD calculations on GPUs," Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pp.1073–1082, May 2014.

Yuttakon Yuttakonkit received the a B.E. degree from Kasetsart University, Thailand in 2011 and a M.E. degree from the Nara Institute of Science and Technology, Japan in 2013. He is currently a doctoral student at the in Graduate School of Information Science, Nara Institute of Science and Technology. His research interests includes low-power processor architecture, high reliability processors, and computational photography. He is a student member of IPSJ and IEICE.

Shinya Takamaeda-Yamazaki received B.E, M.E, and D.E degrees from the Tokyo Institute of Technology in 2009, 2011, and 2014, respectively. From 2011 to 2014, he was a JSPS research fellow (DC1). Since 2014, he has been an assistant professor in the Graduate School of Information Science, Nara Institute of Science and Technology. His research interests include memory systems, FPGA computing, high level synthesis, and processor architecture. He is a member of IEEE and IPSJ.

Yasuhiko Nakashima received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Kyoto University, Japan in 1986, 1988 and 1998 respectively. He was a computer architect in the Computer and System Architecture Department, FUJITSU Limited from 1988 to 1999. From 1999 to 2005, he was an associate professor at the Graduate School of Economics, Kyoto University. Since 2006, he has been a professor at the Graduate School of Information Science, Nara Institute of Science and Technology. His research interests include processor architecture, emulation, CMOS circuit design, and evolutionary computation. He is a member of IEEE CS, ACM and IPSJ.