Data Stream Optimization of Sum of Absolute Differences Algorithm on a Graphics Processing Unit

Tom Pudpai, Tae Kyun Kim, and Charles Liu
Electrical and Computer Engineering, California State University, Los Angeles, California, United States
Email: tompudpai@gmail.com, ttkkus@gmail.com, and cliu@calstatela.edu

***Abstract*— This paper describes the data streaming approaches to performance optimization of the Sum of Absolute Differences (SAD) algorithm on an NVIDIA Graphics Processing Unit (GPU) using the OpenCL programming paradigm. The SAD algorithm forms one of several steps required to implement stereo vision. It creates pixel-based disparity maps from two concurrent images captured by a pair of cameras positioned with a distance in between. The disparity maps can be used to derive depths of objects in the scenes of interest. The massively parallel architecture of a GPU can take advantage of the highly parallelizable SAD algorithm. OpenCL programming framework was chosen to develop the parallel algorithm on the GPU. Performance gains are realized by explicitly mapping data from the slower global memory to the faster shared local memory of the GPU. Local memory is loaded by either a centralized or distributed approach from the OpenCL-defined work-items operating in a workgroup. The resulting performance improvements were discussed based on the architectural features of the GPU and the data streaming approaches used in this research work.**

Keywords - data streaming; Sum of Absolute Differences algorihtm; massive parallel architecture.

I. Introduction

Computer vision is a field of study concerned with extracting information from visual data through computers in a variety of applications, such as robotics, augmented reality, and face detection [1]. Computer vision algorithms typically step through the stages of a vision pipeline. A vision pipeline generally starts from image processing methods to improve results from feature extraction and image analysis. Global and local feature metric extraction form the next stages. Different operations are used on rows or blocks of pixels. In this paper, the SAD algorithm takes place in the local feature metric stage, performing an area operation on the GPU's Single Instruction Multiple Thread (SIMT) architecture.

Advanced Driver Assistance Systems (ADAS) leverage computer vision to increase road safety. One ADAS application is stereo vision, which constructs a three-dimensional image by finding corresponding pixels in image frames from two adjacent cameras [2]. The SAD algorithm is one method to generate the matching costs functions that finds point correspondence in stereo vision. This paper focuses on the use of OpenCL, a generic parallel programming paradigm, to develop the SAD algorithm while utilizing the locality of data reference in the memory hierarchy of a GPU. This research supports the efficient use of restricted memory space in an embedded system for data streaming applications.

Prior research has studied the implementation of the SAD algorithm on different hardware platforms. One study evaluated performance on the FPGA platform with respect to embedded systems [2]. More emphasis was placed on the validity of the algorithm itself, and finding the optimal window size and accuracy over different test image pairs. One of the image pairs was the Venus image sequence, an established stereo vision benchmark that was chosen in this research as well [3]. Another study experimented on the SAD algorithm using an SoPC (System-on-Programmable-Chip) heterogeneous architecture [4]. Their work is similar to ours in that they optimize performance by leveraging on-chip memory and selectively transfer data to off-chip memory. By drawing from the parameters and benchmarks of these previous works, we would like to survey performance speedup of the SAD algorithm on the GPU architecture through optimal data mapping. The rest of this paper is organized as follows. Section II describes the SAD algorithm in relation to computer vision-based ADAS applications. Section III introduces the NVIDIA GPU platform. Section IV describes the OpenCL parallel programming paradigm. Section V described the design and the implementation of the data streaming methodology behind the SAD algorithm implementation and optimization. Section VI described the data streaming approaches. Section VII presents and analyzes the observed results. Section VIII concludes this paper.

II. SAD Algorithm in Advanced Driver Assistance Systems (ADAS)

An ADAS increases driver situational awareness and safety by providing important information to warn the driver of any dangerous events. However, humans are not infallible, and ADAS must eventually advance to take control tasks such as braking or steering, mitigating the errors human drivers make. Eventually, as ADAS applications grow more robust, we can expect fully autonomous vehicles to enter the consumer market.

A variety of sensors enable ADAS applications by providing timely and relevant feedback of the environment. We can roughly categorize these sensors into two categories: time-of-flight and camera [1] (see Figure 1). For front-facing imaging sensors, there are applications available such as lane detection, traffic sign and pedestrian recognition, forward collision warning, and adaptive front-lighting. Imaging sensors that face the rear or side of the vehicle can support ADAS applications, such as parking assistance, rear collision warning, and blind spot detection. Imaging sensors inside the vehicle can even detect occupancy and the alertness of the

Copyright (c) IARIA, 2018.

driver [1]. Detection of vehicles, pedestrians, and traffic signs require substantial computing power. Adding an additional imaging sensor can allow for more accurate and robust detection system by the addition of depth information. The means for extracting depth information from a stereo camera setup, also known as stereo vision. Stereo vision allows 3D information to be extracted from a pair of 2D images taken from adjacent cameras and is an important application of ADAS for vehicles. The fundamental problem with stereo vision analysis is finding the corresponding elements within the image pair. For correct correlation of image pair elements, rectification is required [5]. It ensures that the images are horizontally aligned, allowing for the epipolar curve between each image to be a linear. This means that any algorithm that matches pixels from one image to the next will only need to search horizontally across a row of pixels.

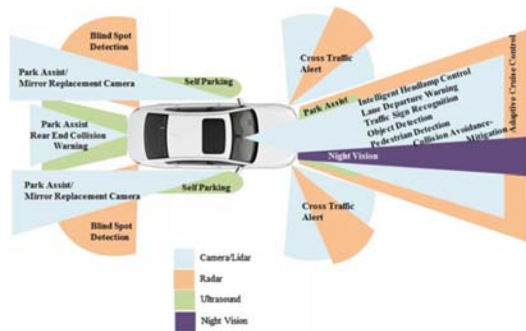

Figure 1. Key applications for ADAS [1].

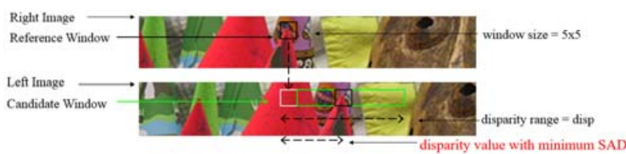

Figure 2. SAD value calculation example; *h x k* = 5 x 5, and *disp*=64.

After rectification, each pixel in one image is matched with a pixel in the other image. Then, a disparity map can be generated, indicating the disparity level of each pixel, to be referenced for acquing depth information. The Sum of Differences (SAD) algorithm is the method chosen to calculate a disparity map in this paper. The benefit of this algorithm is computation efficiency, since the calculations involve primarily addition and subtraction operations. The operational form of the SAD addresses window size and the disparity range because area operations are less computationally costly and depth range is physical limited to the distance between the cameras. For instance, if the disparity range is 64 pixels and the window 5x5, a SAD value may be defined as follows:

$$SAD(i, j, disp) = \sum_{h=-2}^{2} \sum_{k=-2}^{2} |P_R(i+h, j+k) - P_L(i+h, j+k+disp)|$$

Where i and j are the indices of the reference pixel in the right and left images, P_R and P_L respectively, *disp* (the disparity range) is the number of candidate windows that are evaluated in the left image, and h and k define the size of the window. Note that the matching pixel is only searched horizontally after image rectification. Thus, the *disp* is only applied in the second dimension of the left image in the SAD calculation. Figure 2 illustrates the SAD value calculations for matching the tip of a red cone between the right and left images. After the 64 SAD values have been calculated for every pixel from coordinate (i, j) to $(i, j{+}dist)$, the disparity level selected is based on the minimum cost function:

$$Disp(i, j) = \text{Arg Min}(SAD(i, j, disp)), 0 \le disp \le 63$$

Using the Argument Minimum (ArgMin) function, the index of the candidate window with the smallest computed SAD value will be treated as the disparity level for the pixel coordinate (i, j). The disparity range and window size should be scaled based on the parameters of the application where the SAD algorithm is used. The disparity range will depend on the distance between the two cameras, as well as the distance from the camera to the object of interest.

III. NVIDIA GPU PLATFORM

The NVIDIA GeForce 940M graphics card is the primary hardware architecture used to run the SAD algorithm. The GM 108 has three Maxwell Streaming Multiprocessors (SMMs). Figure 3 shows the architecture of an SMM.

Figure 3. Maxwell Streaming Multiprocessor (SMM) block diagram. (excerpted from [6]).

There are 128 cores in each SMM. Each SMM is partitioned into four separate processing blocks, each with its own instruction buffer, scheduler, and 32 cores, as well as a 16,384 x 32-bit register file [6]. There are two L1/texture caches per SMM that act as coalescing buffers for memory accesses. There is also 64 KB of shared memory that can be programmed and allocated by the programmer. Since it is

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-658-3

located on-chip like cache memory, the shared memory can be accessed very quickly. Thus, the explicit streaming of data to shared memory is the focal point of the SAD algorithm optimization for this paper.

IV. Programming Paradigm

OpenCL was used as the Application Program Interface (API) in developing the parallel SAD program on the GPU. It is a heterogeneous programming framework [7]. OpenCL kernels are modeled in a similar manner to Single Program Multiple Threads (SPMT), where parallel threads (i.e., work-items) execute instances of the kernel to map effectively on both scalar and vector hardware. The OpenCL specification can be divided into four models: *Platform model, Execution model, Programming model,* and *Memory model [8].* The *Platform model* specifies that there is one host processor that coordinates execution of kernels, and that there are one or more device processors that actually execute the kernels. Each device is modeled as a group of compute units, which are further divided into processing elements where each element can execute instances of kernels. The *Execution model* defines how the OpenCL environment is configured by the host, and how the host may direct the devices to perform work. The *Programming model* defines how concurrency is mapped to physical hardware. Each unit of concurrent execution is defined as a work-item, which executes the kernel function body. The work-items are indexed in an n-dimensional range, also known as NDRange. To achieve scalability, the work-items of an NDRange can be divided into equally-sized workgroups. Synchronization of work-items is only possible within workgroups (see Figure 4). The workgroup and global work-item size dimensions are specified by the programmer and must be a power of two number. Also, the global work-item size must be evenly divisible by the workgroup size [8].

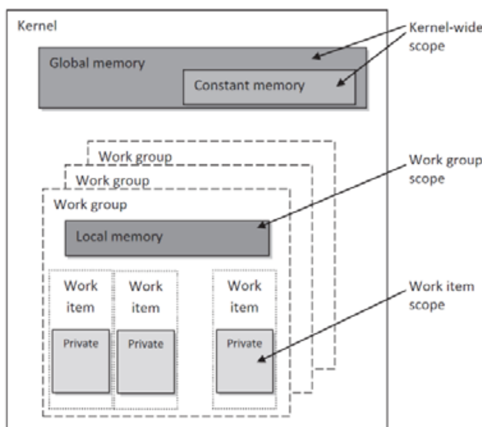

Figure 4. OpenCL Programming and Memory models [5].

The *Memory model* defines memory object types, and the abstract memory hierarchy that kernels use regardless of actual underlying hardware architecture. Memory in OpenCL is divided into host memory and device memory [8]. Device memory is divided into global memory, local memory, private memory, and constant memory (see Figure 4). Global memory can be read from or written to by all work-items running on the device. Data transferred to or from the host will reside in global memory. Reads and writes may be implicitly cached depending on the capabilities of the device [7]. Local memory is shared by work-items in a workgroup only. It is typically mapped to on-chip memory that has shorter latency and higher bandwidth than global memory. Private memory is visible only within a work-item. Constant memory is a region of global memory that remains constant during kernel execution.

The memory model of OpenCL is well suited for NVIDIA GPUs. Each core running a thread, or OpenCL work-item, contains dedicated private memory. All workgroups can communicate through global memory located in off-chip GPU memory. SMMs have dedicated shared memory for communication between work-items in a workgroup, which fits the role of OpenCL's local memory. Accessing this shared memory is fast as long as there are no bank conflicts between threads [9]. Shared memory is divided into equally sized memory banks, which can all be accessed simultaneously. If there are multiple requests to the same bank, the requests become sequential, incurring memory access delays. Therefore, for maximum performance, bank conflicts should be minimized by considering how the memory addresses are mapped.

V. Design and Implementation

In the OpenCL Platform Model, the host sends commands to the device to transfer data between host and device memories, as well as to execute the parallel device code. The host (an Intel Core i5) executes serial code and is typically a CPU. The host is responsible for setting up the execution pathway to and from the device, and requires a lengthy setup process which begins by identifying the platform and device. Memory buffers must be created to link objects on the host to objects in the kernels executed on the device. For this research, memory buffers are needed for the left and right input image values, the output SAD values, the image dimensions, the padded image dimensions (to round up to the closest power of 2 number in each image dimension as explained below), disparity level, window dimensions, work item dimensions, and conditional values.

The device is responsible for execution of the kernel as directed by the host. Initially, the input images, disparity output, and other kernel parameters defined in the previous section are transferred from the host memory and allocated to the global memory of the device. In this paper, the images used are the Venus pair, used in several benchmarks amongst stereo vision researchers. Given the 384 pixels x 434 pixels image size, there are 162,222 Disparity Levels to be calculated based on the SAD algorithm (see Section II). Each one is executed on a work-item. Because of the size restriction by OpenCL, we must round up the image dimensions to the nearest power of 2 in order to process every pixel of the image pair. Thus, the image values of L and R are padded with values

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-658-3

of 0 to reach dimensions of 512 x 512. The partitioning of work-items into workgroups is determined by the OpenCL local-item size and global-item size dimensions. In this NVIDIA GeForce 940M GPU architecture, each work-item from OpenCL is operated on a GPU core. Each workgroup is operated on an SMM with 128 cores. Thus, 128 work-items can run in parallel. Since this GPU has 3 SMMs, a total of 3 * 128 = 384 work-items can run in parallel. If the GPU were to run at maximum occupancy, there would be $\lceil 162222/384 \rceil = 423$ iterations of the SAD algorithm needed.

To differentiate and track each work-item, the OpenCL API function get_global_id() is used to return its unique global ID value [8]. This is important because the instances of the kernel operating on SAD values of the image edges must be treated differently. In this paper, without losing the generality of the parallel algorithm, we use a common window size with 5 pixels x 5 pixels for the SAD algorithm for performance analysis. A reference window in an image compares to 64 iterations of candidate windows in the counterpart image. Note that the SAD values on the border cannot be computed because the 5 x 5 windows will be incomplete. Thus, the SAD values cannot be computed for kernels two pixels within each border. In the device kernel code, this padding is implemented through a conditional statement with a reference to the global ID of the kernel to avoid the incomplete calculations of such close-to-border SAD values. Upon finding the minimum SAD value for all 64 candidate windows, the corresponding disparity level must be saved to the disparity map output. The output matrix is stored as an integer array in global memory.

The performance of the SAD algorithm can be first enhanced through loop unrolling. Loop unrolling involves the rewriting of loops into a repeated sequence of similar independent statements. This helps eliminate the loop overhead and also hides stalls due to data dependencies [10]. The original implementation of the SAD algorithm in this paper consists of a nested *for loop* that increments the kernel's SAD value a total of 25 times, one for each pixel in the 5 x 5 window. The disadvantage to this approach is much lengthier code, which is particularly harmful to embedded systems with limited instruction memory.

The other important factor to affect performance is the workgroup size; i.e., the number of work-items defined in a workgroup. In our design, the workgroup size varies from 32 to 512. Based on the feature of the GPU hardware, there are 128 x 3 = 384 cores. 512 is that number's next power of two value. Thus, a workgroup size greater 512 is not considered due to the mismatch to the hardware.

VI. Data Streaming Optimization

The first SAD algorithm in this paper was implemented to access all data from the GPU's global memory. Global memory is visible to all of the Streaming Multiprocessors in the GPU, but is located off-chip, so accesses to global memory incur heavy delays. As addressed in Section III, The OPENCL local memory is mapped to the SMM's shared memory, which is shared by all of the cores in that single SMM, and is stored on-chip (see Figure 3). By taking advantage of this local memory and the mapping scheme for utilizing the spatial and temporal localities of data, significant speedup can be achieved for the SAD algorithm.

Datatype optimizations are possible through OpenCL. As mentioned in Section III, memory copying incurs performance penalties because bandwidth and power wasted on data transfer. We must consider the input format of our algorithm, which is made up of pixel intensity values between 0 and 255. This means that only an 8-bit unsigned integer is required to store the input value. Previously, we have used 32-bit signed integers to transfer from the CPU host to the GPU device. OpenCL does not provide support for 8-bit unsigned integer types, but it does allow for an 8-bit unsigned char type. By typecasting the 32-bit integer pixel intensity input values to type unsigned char, we can reduce the memory copied to the GPU by 75%. This produces a noticeable decrease in execution time.

A. Centralized Memory Access

The first implementation of the data streaming optimization requires that the first work-item in the first row of a work-group to process its kernel will populate local memory with the necessary pixel values needed by the workgroup row. This is considered the centralized memory access approach to data streaming optimization. The centralized approach of having one work-item load local memory for its workgroup is depicted in Figure 5.

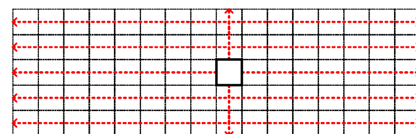

Figure 5. Centralized loading approach visualization.

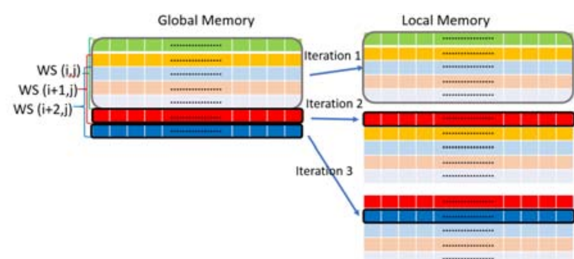

Figure 6. Round-robin local memory loading.

Figure 6 shows the data streaming from the global memory to the local memory. The working set, *WS(i, j)*, shows the amount of memory needed to determine *Disp(i,j)*. In Iteration 1, 5 rows of global memory are loaded to local memory. In subsequent iterations (for determining *Disp(i+1, j)*, *Disp(i+2, j)*, etc.), only 1 new row needs to be loaded and to replace an existing row in the local memory in a round-robin fashion to fulfill the local memory accesses to their corresponding working sets. This approach reduces the data

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-658-3

accesses by utilizing the spatial and temporal localities in local memory.

Modifications to the host-side code are required for implementation of local memory optimization. The size of local memory allocated on the device must be specified by the host code. For the NVIDIA GeForce 940M GPU, the local memory capacity is 49,152 bytes. For this implementation, using a 5 x 512 local memory size, where each value is represented by a 4-byte integer, leads to 5 x 512 x 4 = 10,240 bytes allocated in local memory. Both left and right images require their own local memory allocations, leading to 20,480 bytes allocated total. This local memory allocation is explicitly declared when setting the kernel argument for the device code kernel. Normally, this kernel argument is linked to a memory buffer previously defined in the host code. However, data in local memory is private to the workgroup in the device. Thus, data is never read from or written to local memory from the host directly.

Another modification needed for the host code is the declaration of a Boolean array named rowDone, which keeps track of completed rows of work-items in each workgroup. The size of this array is equal to the height of the padded image times the number of workgroups along the width of the original image. The implementation of this array allows work-items in consequent rows to check the status of the work-items in the previous rows prior to completing execution. This array must be declared as a readable and writable memory buffer since it must be read from and written to by different work-items.

The first implementation of this data streaming optimization requires that every work-item populate local memory with the relevant data for its workgroup. Each work-item begins by defining boundaries for the data that must be loaded to local memory. The work-items in the first row of the workgroup will load local memory first and then perform the SAD algorithm. The following row will wait until this previous row has finished execution, and will then replace one row of local memory with the next row of data from global memory. When the last work-item of a row has finished execution, it will set rowDone to "true" for its corresponding row and workgroup. This is possible because in OpenCL, work-items execute in order along rows of work-items in a workgroup.

B. Distributed Memory Access

The Centralized Memory Access approach will introduce increased workload to the first work-item as the number of work-items in a workgroup increases. This is due to the pre-load of a larger number of working sets. Thus, it may eventually cause workload imbalance among the work-items. In this paper, the second approach to data streaming optimization is distributed memory access. We attempt to distribute the task of loading to local memory equally among all of the work items. In this manner, the work is divided evenly within each workgroup, and no work-items are left idle. Figure 7 depicts this process of distributed loading for one row of work items. In Iteration 1, each work-item loads 5 pixels, where the center pixel has the same image coordinates as the global ID of that work-item. Then, in the subsequent iterations, the pixels from the next rows will be read by the corresponding work-item and be located to the local memory buffer in a round-robin fashion. The mapping is the same as shown in Figure 6.

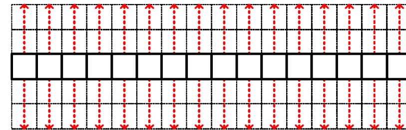

Figure 7. Distributed loading approach visualization.

Conceptually, this implementation is much simpler than the centralized approach. Previously, the centralized approach involved multiple if-else statements to check the row position and whether the previous row of work-items completed execution. Without these conditional statements, the distributed approach saves execution time

For implementation of this distributed approach to the data streaming optimization, modifications are made to the kernel code alone. The scenario in Figure 7 where each work-item loads exactly the same number of pixel values from to local memory is ideal, but not feasible. The work-items are executing in parallel, but the latency to load the pixels across the boundaries of local memory is significantly higher than it of the others due to the lack of spatial locality of accessing their local memories across the boundaries. This may result in some work-items attempting to calculate SAD disparity values before these lagging work items have completed loading the required values. Therefore, they must be synchronized with a barrier. All work-items in a workgroup must execute the OpenCL function "barrier(CLK_LOCAL_MEM_FENCE)" before they can proceed, and the CLK_LOCAL_MEM_FENCE flag ensures that local memory accesses are visible to all work-items in the workgroup [8].

VII. Experimental Results

The performance of the three implementations 1) global (the first implementation with global memory access) 2) centralized local (Section VI A), and 3) distributed local (Section VI B) are compared.

There is a trend of declining execution time as the workgroup size increases in Figure 8. It can be explained as the better mapping of the parallel SAD algorithm to the GPU hardware. A larger workgroup will allow more work-items to share local memory, and hence, have better temporal and spatial locality in memory accesses. As expected, the SAD algorithm with global memory access had the worst performance as the workgroup size was set smaller than 256. At the largest workgroup size of 512, the two aforementioned approaches have similar execution times at 18.03 ms, and 18.01 ms, respectively. This is explained as the increase of

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-658-3

overhead due to the workload imbalance on the first work-item, which is responsible for pre-loading all working sets to the local memory for the entire workgroup. In contrast, the distributed approach to the local memory data streaming optimization remains consistently faster than the others, ending up at 4.88 ms for the same workgroup size of 512. 4.88 ms for one disparity calculation would lead to 1 / 0.00488 ≅ 205 frames per second, without taking into consideration the overhead between frames.

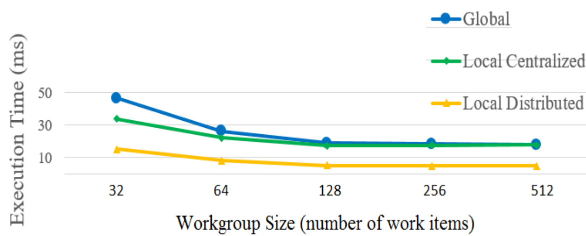

Figure 8. Comparison of SAD algorithm Performance across Different Optimizations.

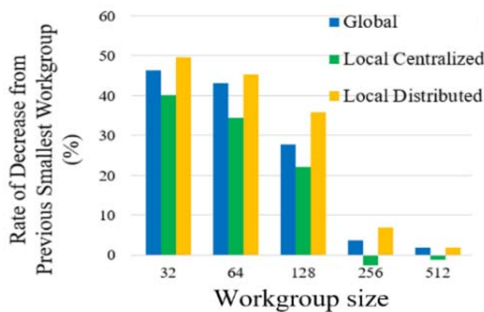

Figure 9. Comparison of Rate of Decrease from Previous Smallest Workgroup for SAD Algorithm Across Different Optimizations.

As the workgroup size gets larger, the rate of decrease in execution time generally decreases, as depicted in Figure 9. The centralized local approach consistently decreases the rate of decrease until it achieves a negative rate from workgroup size 128 to 256 and workgroup size 256 to 512. A negative rate of decrease means that the execution time actually increased. The other approaches have a consistent rate of decrease between 40 and 50%, until reaching a workgroup size of 128. For the three approaches described previously, the rate of decrease is diminished but still positive when transitioning from workgroup sizes of 128 and above. Performance is expected to peak at workgroup size 128 and drop off as the workgroup size increases, but performance continues to increase. These results can be partially attributed to the implicit use of spatial/temporal locality of memory accesses stored in caches by OpenCL. The continuing performance gain may also be explained by the number of kernels queued to an SMM exceeding the number of cores available, leading to a queuing delay. Each SMM has 4 instruction buffers that delegate instructions to their respective cores, and they are loaded with kernel instances each time a workgroup is executing. Larger workgroup sizes mean fewer workgroups, and fewer times the instruction buffers must be loaded.

VIII. CONCLUSION

This paper has shown that the SAD algorithm can be optimized on a GPU platform through OpenCL by explicit programming of local memory data loading and implicit data caching. Code optimizations and explicit caching of global memory have been observed to increase performance. Switching from a centralized approach to a distributed approach to local memory loading further improves performance. This work can be applied to embedded systems running ADAS applications where immediate distance calculation of objects is crucial and life-saving. With a maximum disparity map calculation rate of roughly 205 frames per second on a CPU-GPU heterogeneous environment, this algorithm optimization will surely make a beneficial impact when implemented on real-time embedded systems in automobiles. The work can scale to GPUs with more cores, and to higher resolution images. The code would be very similar in either case. In the future, we hope to continue the distributed memory access optimization approach by parallelizing the loading in a vertical fashion for each workgroup, which will enable us to compute multiple disparity values from the same kernel. We would also like to port this code to an embedded platform to see if the real-time performance gain will carry over as suspected.

REFERENCES

[1] B. Kisačanin and M. Gelautz, Eds., Advances in Embedded Computer Vision, Springer International Publishing, 2014.
[2] Citron, C. (2014). “Stereo vision system module for low-cost FPGAs for autonomous mobile robots”. doi:10.15368/theses.2014.149
[3] D. Scharstein and R. Szeliski, "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," International Journal of Computer Vision, vol. 47, pp. 7-42, 2002. doi: 10.1109/SMBV.2001.988771,
[4] X. Zhang and Z. Chen, "SAD-Based Stereo Vision Machine on a System-on-Programmable-Chip (SoPC)," Sensors, no. 13, pp. 3014-3027, 2013. doi: 10.3390/s130303014.
[5] K. Konolige, “Small Vision Systems: Hardware and Implementation”, Proceedings of the 8th International Symposium in Robotic Research, pp. 203–212, 1997.
[6] NVIDIA, "NVIDIA GeForce GTX 750 Ti Whitepaper," 2014.
[7] D. Kaeli, M. Perhaad, D. Schaa, and D. P. Zhang, Heterogeneous Computing with OpenCL 2.0, Morgan Kaufmann, 2015.
[8] Khronos Group, "The OpenCL Specification Version 2.0," 2015.
[9] NVIDIA, "OpenCL Best Practices Guide," 2011.
[10] A. Nicolau, "Loop quantization : unwinding for fine-grain parallelism exploitation," Cornell University, Dept. of Computer Science, Ithaca, N.Y., 1985.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-658-3