International Journal of Computer Applications (0975 – 8887)
Volume 164 – No 3, April 2017

Efficient Dynamic Multiple GPGPU Layer for OpenCV

Afshan Jafri
Information Technology Department
College of Computer and Information Sciences,
King Saud University,
P.O. Box 22452, Riyadh,
11495, Saudi Arabia

ABSTRACT

General purpose graphic processing unit (GPGPU) provides high performance resource for computing. CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) permit writing of parallel computing programs that utilize multiple central processing units (CPU) and GPGPUs. The image processing library, OpenCV (Open Source Computer Vision library), may benefit greatly from parallel use of multiple GPGPUs, however, its CUDA implementation is restricted to benefiting from a single GPGPU only. This research develops an abstraction layer above OpenCV single GPU module that enables multiple GPUs for single instruction multiple data (SIMD) architecture. This approach has a controller/parent thread which generates various worker threads to operate on several GPU devices, to handle balancing of work load on GPUs, as the task allocation is dynamic for any number of GPUs. The experiments on running bilateral filtering, color to gray conversion, fast Fourier transform, and convolution on homogeneous and heterogeneous sized images of scenery, objects, and faces, indicate that: (1) threading reduces computation time by half of sequential operation for GPU; (2) tuned static load balanced GPU threading reduces computation time by up to a fourth when compared to CPU threading; (3) performance of dynamic load balancing approaches that of manually iteratively balanced static operation.

General Terms

High performance computing, parallel computing, scientific programming, computer vision

Keywords

GPGPU, OpenCV, SIMD, CUDA, OpenCL, Multiple GPU, Load Balancing, Threading.

1. INTRODUCTION

General processing units (GPU) have been used for graphics processing for the last three decades. With increased power, programmability and flexibility, GPU functionality has been extended from graphics only processing to general purpose GPGPU to provide a flexible, high performance resource for scientific processing applications [1]. Multiple GPGPU's provide substantial improvements in performance with suitable programming frameworks [2].

The standardized programming frameworks CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) permit writing of general purpose parallel computing programs that utilize multiple central processing units (CPU) and GPGPUs [3]. CUDA is specific to GPU built by NVIDIA, whereas OpenCL is for heterogeneous devices supported by consortium. As CUDA is constructed for a homogeneous environment, it outperforms the portable OpenCL for a standard programming, memory models and optimization options [4, 5].

A toolkit that can benefit greatly from parallel use of multiple GPGPUs is the image processing library, OpenCV (Open Source Computer Vision library). It is collection of functions written in C/C++ to provide simple-to-use real time computer vision infrastructure [6].

With expansion of GPU to general purpose processing, OpenCV built their library to support their function to run on only a single GPU to utilize the high performance computing of GPGPU [7]. In high performance computation, it is often required to operate few functions on large chunk of data as Single Instruction Multiple Data (SIMD). These requirements typically get handled with parallel processing methodology. Although GPU provides parallel computing architecture to process tasks fast in parallel but with only single GPU support of OpenCV. This provides the bottleneck in high performance OpenCV computation.

This paper develops an abstraction layer above OpenCV single GPU module that enables multiple GPUs for single instruction multiple data architecture, as illustrated in Figure 1. The goal is to enable all single GPU OpenCV functions to run on multiple GPU concurrently to handle single instruction multiple data scenario. This objective is accomplished by building an abstraction layer on OpenCV single GPU module to enable multiple GPU implementation of OpenCV.

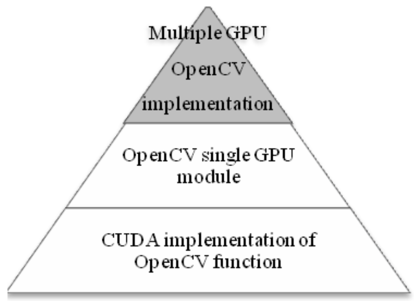

Figure 1: Hierarchy/abstraction of the layer implementation

The construction of the software layer is in CUDA and is applied to functions in OpenCV because of the large OpenCV community of developers and users that can be directly impacted by this work and the optimized CUDA environment for GPU [8,9]. Appendix A provides a sample of the code developed for multiple GPU abstraction layer.

The paper is organized as follows: Section 2 describes the methodology; Section 3 presents the algorithms developed; Section 4 explains the experiment setup to test the performance of the algorithms and presents the results; Section 5 provides the conclusions.

2. METHODOLOGY

One of the methods in parallel processing is multiple threading [10,11]. In order to process tasks in parallel, multiple threads are generated. Each thread deals with its own copy of data and function to be executed on that data. The approach in this paper uses multiple threads on OpenCV single GPU implementation to handle SIMD scenario. This maximizes the usage of processing power.

This approach, as depicted in Figure 2, is based on having a controller/parent thread that generates various worker threads to operate on all available GPU devices. Each worker thread is given data and task to execute on the data using particular device. Each thread takes control of particular device, performs computation and reports back the result to the main thread. Parallel processing is achieved with multiple threading as worker threads operate concurrently on their own with no interdependence. Controller thread maintains the task allocation and thread synchronization of worker (child) threads before all of them terminate.

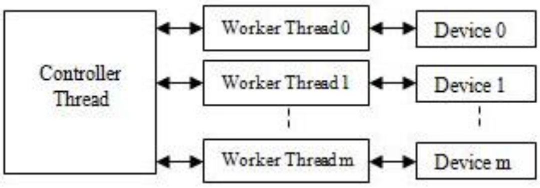

Figure 2: Framework of our methodology

3. LOAD BALANCING

The framework conducts load balancing to divide the workload (data and processing) among the GPGPU devices according to their capabilities or processing power, in order to optimize the overall performance. Allocation of the data/load for execution on the various devices requires algorithms for distribution of data and load [12]. This paper develops two methods of load balancing: static and dynamic. In the static version, load distribution is pre-decided for each GPGPU device, whereas in dynamic version, load/data is allocated to the devices during the run time only (dynamically).

3.1 Static Balancing Algorithm

The static balancing algorithm consists of four parts: planner, controller thread, worker thread, and synchronization & termination as illustrated in Figure 3. The following paragraphs explain the process flow for the static version implementation of Multiple GPGPU OpenCV.

The planner phase detects the devices available and optimizes task load for each device according to computational power of the device. Experiments indicate that device computational capability depends mainly on two factors: number of cores (NC), and device clock rate (CR).

The load distribution is calculated as follows for two devices, Dev_1 and Dev_2, and a total load of TL. Let NC_1 and NC_2 denote the number of cores available in the two devices, and CR_1 & CR_2 represent the device clock rate of Dev_1 and Dev_2 respectively. Load L_1 for Dev_1 is calculated as follows:

$$L_1 = \left[\alpha \frac{NC_1}{NC_1 + NC_2} + (1-\alpha) \frac{CR_1}{CR_1 + CR_2} \right] TL$$

Here α is a tuning parameter that allows the user to decide on the relative weight of NC with respect to CR. A default value of 0.5 give equal weight, a value above 0.5 favors number of cores (NC) and a value below 0.5 gives preference to the clock rate (CR). A user manually varies α around default value based on GPGPU specification for optimal performance of heterogeneous data processing.

Once load distribution is decided, worker threads are created dynamically in the controller thread phase with one thread per device. The main controller thread then assigns pre-decided load/data and functions (single GPGPU OpenCV functions to operate on the data) to each worker thread.

In the third phase, each worker thread takes control and runs the user defined function on particular GPGPU device. When worker threads are done with task assigned, controller thread synchronizes and terminates them in the fourth phase.

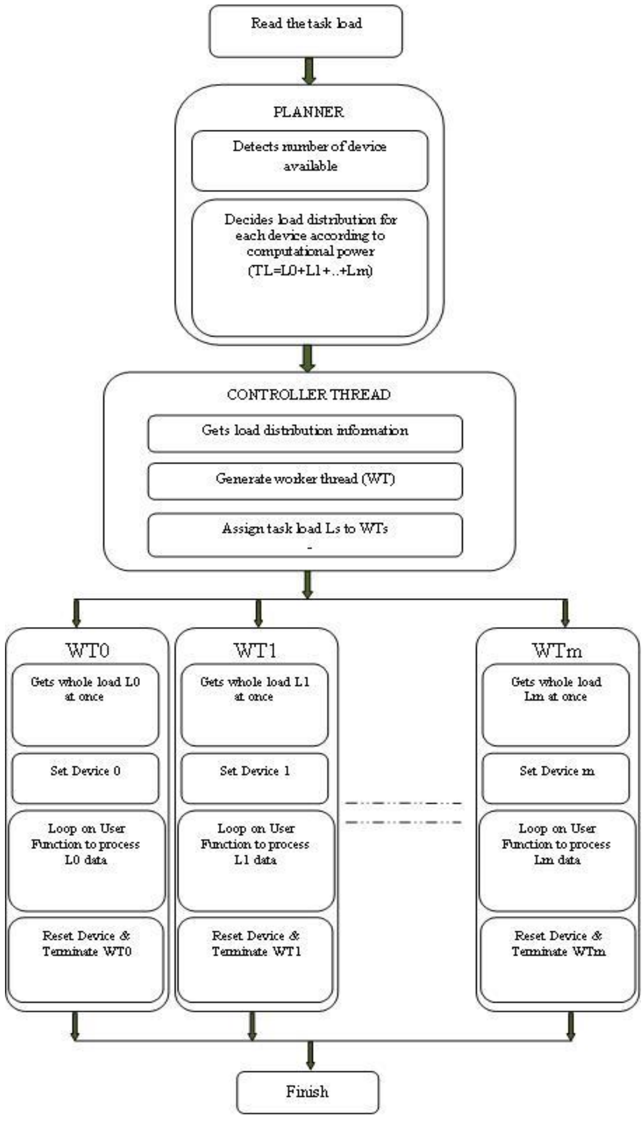

Figure 3: Static load balancing

3.2 Dynamic Balancing Algorithm

The dynamic load balancing algorithm distributes load on devices during runtime rather than pre-allocating the load before execution [13]. It automatically queues tasks according to the busy status of the device, as illustrated in Figure 4.

The following paragraphs explain the process for dynamic load balancing on multiple GPUs. The algorithm reads user defined directory for the data files, makes a list of filenames, and conducts load allocation in three phases: controller thread, worker thread, and synchronization / termination.

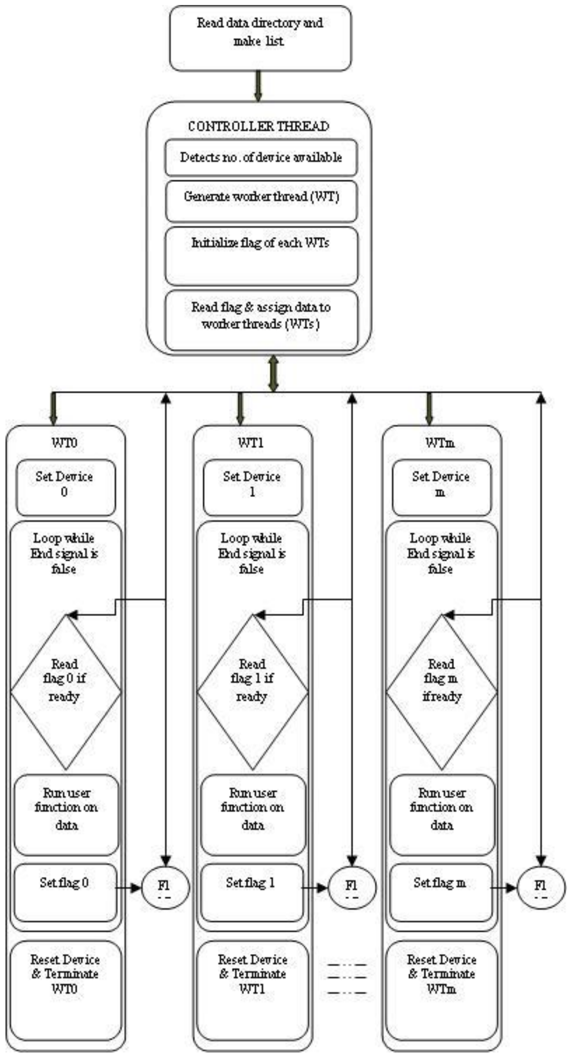

Figure 4: Dynamic load balancing

The controller thread phase detects the number of devices available, forms load queue, and dynamically creates worker threads (one thread per device). Each thread gets functions (single GPU OpenCV functions) and one data at a time to work on.

A thread has a flag to report the status of execution: busy or ready, which is initialized by the controller thread to ready. The thread operator loops on the data list to process and reads the flag of each worker thread (one worker thread controls one device). If the device is ready, it assigns (another) data from the load queue to that thread to work on and writes flag as busy; otherwise waits for some time. The thread operator gives the 'end' signal to terminate execution.

The worker thread takes data from the Controller thread and reads its flag. If the worker thread is flagged as busy by the Controller thread, the task (user defined function with data) is assigned to the device; otherwise it waits. It gives the thread kill command on receiving the end signal from the Controller thread. Finally, in the third phase, when the load queue is empty, the controller thread synchronizes and terminates worker thread.

4. EXPERIMENTATION AND RESULTS

Experimental design is conducted to investigate the performance of the following: (E1) threading and sequential computation; (E2) Static GPU threading and CPU threading; and (E3) static and dynamic load balancing approach. A variety of functions and data are used such as bilateral filtering, color to gray conversion, fast Fourier transform, and convolution; homogeneous and heterogeneous sized images of scenery, objects, and faces.

The platform used for experimentation has Intel Xeon CPU E5-2603 0 1.8GHz as CPU and two devices as GPU. Device 0 is "Tesla C2075", and Device 1 is "Quadro 2000". Details of the hardware are provided in Appendix B. The software used in the experiments is OpenCV 2.4.8, Microsoft visual studio 2008, and Pthread –POSIX 1003.1c (dynamic platform independent thread management). The data used for the tests are: test data from OpenCV extra from git-hub [14], Caltech256 database [15], and Standard test images from imageprocessing.com [16].

4.1 Threading and Sequential Operation

This study investigates the effects of threading in relation to sequential operation in both GPU and CPU to form a baseline for performance. The experiments utilize bilateral filtering function of thirty high resolution image (4096x3072) [14,15,16]. Table 1 presents computational time for: (1) GPU threading, (2) GPU sequential, (3) GPU threading - load distributed (4) CPU threading, (5) CPU sequential. The results indicate that GPU sequential computation is four times faster than CPU sequential. The computational time for GPU is halved using threading when compared to sequential. In contrast, CPU threading is less effective in improving performance. The conclusion of the study is that using multiple threading in multiple GPUs scenario with balanced work load as per GPU capabilities provides the fastest computation.

Table 1 Comparison between threading and sequential operation in GPU and CPU

Task performed: Bilateral filtering of high resolution image (4096x3072)
Number of images to process: 30

Approach	Device (GPUs)	Thread number	No. images (work load)	Computational time (secs)
GPU Threading	Device 0 (Tesla)	Thread 0	17	**56.6**
	Device 1 (Quadro)	Thread 1	13	
GPU Sequential	Device 0	NA	30	91.52
	Device 1	NA	30	117.72
GPU Sequential (load distributed)	Device 0	NA	17	106.98
	Device 1	NA	13	
CPU Threading	CPU 1	Thread 0	15	411.109
	CPU 2	Thread 1	15	
CPU Sequential	CPU	NA	30	479.493

4.2 Static Load Balanced GPU and CPU Threading

This section compares the performance of tuned static load balanced GPU threading and CPU threading. Three tasks are performed on 30 to 50 high resolution images (4096x3072): (1) bilateral filtering, (2) color to gray conversion followed by

2D filtering, (3) Fast Fourier Transformation of high resolution grayscale image. The OpenCV functions used are bilateralFilter; cvtColor, filter2D; and dft, merge, split, magnitude, log, and normalize. Table 2 presents computational time for static load balanced GPU threading and CPU threading for two devices. In all three tasks, GPU threading significantly outperforms CPU threading, with factors ranging from more than half to an eighth.

Table 2 Computation time of tuned static load balanced GPU threading and CPU threading

TASK 1:
Task Performed: Bilateral filtering of high resolution image (4096x3072)
Parameters: kernel size=-1, sigma color=50, sigma space= 7
OpenCV functions used: bilateralFilter
Number of images to process: 30 (homogeneous size)

Approach	Device (GPUs)	Thread number	# images (work load)	Computational time (secs)
GPU Threading	Device 0 (Tesla)	Thread 0	17	56.6
	Device 1 (Quadro)	Thread 1	13	
CPU Threading	CPU 1	Thread 0	15	411.109
	CPU 2	Thread 1	15	

TASK 2:
Task Performed: Color to Gray conversion + user defined 2D filtering of high resolution image (4096x3072)
Filter is 16x16 ones with normalization factor of 16*16
OpenCV functions used: cvtColor, filter2D
Number of images to process: 50 (homogeneous size)

Approach	Device (GPUs)	Thread number	# images (work load)	Computational time (secs)
GPU Threading	Device 0 (Tesla)	Thread 0	28	58.297
	Device 1 (Quadro)	Thread 1	22	
CPU Threading	CPU 1	Thread 0	25	139.202
	CPU 2	Thread 1	25	

TASK 3:
Task Performed: Fast Fourier Transformation of high resolution grayscale image (4096x3072)
OpenCV functions used: dft, merge, split, magnitude, log, normalize
Number of images to process: 50(homogeneous size)

Approach	Device (GPUs)	Thread number	# images (work load)	Computational time (secs)
GPU Threading	Device 0 (Tesla)	Thread 0	28	94.376
	Device 1 (Quadro)	Thread 1	23	
CPU Threading	CPU 1	Thread 0	25	220.448
	CPU 2	Thread 1	25	

4.3 Static and Dynamic Load Balancing

Experiments are conducted to compare computation time for three load balancing methods: non-tuned static; optimized static using manual tuning; and dynamic. Naturally, non-tuned static is expected to have worst performance and the manually optimized static load balancing to have the best performance. The objective is to investigate how close is the practical dynamic load balancing compared to the cumbersome optimized static version.

Computation is investigated in the context of three tasks: (1) bilateral filtering (OpenCV function is bilateralFilter); (2) Fourier transformation (OpenCV functions being dft, merge, split, magnitude, log, normalize); (3) color-to-gray conversion followed by user defined 2D filtering (OpenCV functions being cvtColor, filter2D). For each task, four sets of images are used: (1) scenery consisting of 100 heterogeneous sized images; (2) objects (e.g. guns and shoes) of 27 heterogeneous sized images; (3) thirty facial images of heterogeneous sizes; (4) ten gray scale heterogeneous sized images.

Table 3.1 Load balancing using Static and Dynamic approach - Task 1: Bilateral Filtering

Images 1: Bilateral filtering of SCENERY images
Number of images to process: 100 (heterogeneous size)
Parameters: kernel size=-1, sigma color=50 and sigma space= 7
OpenCV functions used: bilateralFilter

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	64.894
		Device 1 (Quadro)	Thread 1	
Static (non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	67.361
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	65.458
		Device 1 (Quadro)	Thread 1	

Images 2: Bilateral filtering of OBJECT (guns, shoes, etc.,) images
Number of images to process: 27 (heterogeneous size)
Parameters: kernel size=-1, sigma color=50 and sigma space= 7
OpenCV functions used: bilateralFilter

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	7.791
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	8.861
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	8.049
		Device 1 (Quadro)	Thread 1	

Images 3: Bilateral filtering of FACIAL images
Number of images to process: 30 (heterogeneous size)
Parameters: kernel size=-1, sigma color=50 and sigma space= 7
OpenCV functions used: bilateralFilter

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	7.659
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	8.052
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	7.644
		Device 1 (Quadro)	Thread 1	

Images 4: Bilateral filtering of GRAYSCALE images
Number of images to process: 10 (heterogeneous size)
Parameters: kernel size=-1, sigma color=50 and sigma space= 7
OpenCV functions used: bilateralFilter

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static	GPU	Device 0	Thread	7.525

(manually tuned)	Threading	(Tesla)	0	
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	8.362
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	7.66
		Device 1 (Quadro)	Thread 1	

Table 3.2 Load balancing using Static and Dynamic approach - Task 2: Fourier Transformation

Images 1: Fourier Transformation of SCENERY images
Number of images to process: 100 (heterogeneous size)
OpenCV functions used: dft, merge, split, magnitude, log, normalize

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	86.471
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	95.769
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	87.067
		Device 1 (Quadro)	Thread 1	

Images 2: Fourier Transformation of OBJECT (guns, shoes, etc.,) images
Number of images to process: 27 (heterogeneous size)
OpenCV functions used: dft, merge, split, magnitude, log, normalize

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	7.8
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	9.142
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	8.331
		Device 1 (Quadro)	Thread 1	

Images 3: Fourier Transformation of FACIAL images
Number of images to process: 30 (heterogeneous size)
OpenCV functions used: dft, merge, split, magnitude, log, normalize

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	7.659
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	9.329
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	8.05
		Device 1 (Quadro)	Thread 1	

Images 4: Fourier Transform of GRAYSCALE images
Number of images to process: 10 (heterogeneous size)
OpenCV functions used: dft, merge, split, magnitude, log, normalize

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static	GPU	Device 0	Thread	7.488

(manually tuned)	Threading	(Tesla)	0	
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	8.767
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	7.724
		Device 1 (Quadro)	Thread 1	

Table 3.3 Load balancing using Static and Dynamic approach - Task 3: Color-To-Gray Conversion

Images 1: Color to Gray conversion + user defined 2D filtering of SCENERY images
Number of images to process: 100 (heterogeneous size)
OpenCV functions used: cvtColor, filter2D
User defined 2DFilter is 16x16 ones with normalization factor of 16*16

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	45.817
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	50.653
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	46.52
		Device 1 (Quadro)	Thread 1	

Images 2: Color to Gray conversion + user defined 2D filtering of OBJECT (guns, shoes, etc) images
Number of images to process: 27 (heterogeneous size)
OpenCV functions used: cvtColor, filter2D
User defined 2DFilter is 16x16 ones with normalization factor of 16*16

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	7.785
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	8.705
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	7.925
		Device 1 (Quadro)	Thread 1	

Images 3: Color to Gray conversion + user defined 2D filtering of FACIAL images
Number of images to process: 30 (heterogeneous size)
OpenCV functions used: cvtColor, filter2D
User defined 2DFilter is 16x16 ones with normalization factor of 16*16

Load Balancing	Approach	Device (GPUs)	Thread number	Computational time (secs)
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	7.878
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	8.829
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	7.927
		Device 1	Thread	

		(Quadro)	1	
Images 4: Color to Gray conversion + user defined 2D filtering of GRAYSCALE images Number of images to process: 10 (heterogeneous size) OpenCV functions used: cvtColor, filter2D User defined 2DFilter is 16x16 ones with normalization factor of 16*16				
Load Balancing	**Approach**	**Device (GPUs)**	**Thread number**	**Computational time (secs)**
Static (manually tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	7.145
		Device 1 (Quadro)	Thread 1	
Static (Non-tuned)	GPU Threading	Device 0 (Tesla)	Thread 0	8.135
		Device 1 (Quadro)	Thread 1	
Dynamic	GPU Threading	Device 0 (Tesla)	Thread 0	7.348
		Device 1 (Quadro)	Thread 1	

Tables 3.1, 3.2, and 3.3 present the computational times for each of the three tasks with sub-tables showing the results for each of the four sets of images. The results clearly indicate consistently and in all tasks that dynamic load balancing is closer in efficiency to manually tuned static balancing than to non-tuned static load balancing. In many case, computational time of dynamic load balancing is comparable to the optimized manually tuned static load balancing. This is in addition to dynamic balancing being automatic compared to fine tuning which is manual, iterative, and cumbersome.

5. CONCLUSION

This paper has constructed an abstraction layer above OpenCV single GPU module to enable multiple GPUs for SIMD architecture, and developed static and dynamic load balancing mechanisms. Multiple experiments are conducted for a variety of tasks and images to compare computational performance of various schemes. The conclusions are as follows:

(1) Balanced workload on GPU provides the fastest computation compared to CPU threading

(2) Static load balancing works well and makes full utilization of GPUs only if the data of interest is homogeneous in nature (of same size and type). However, it is cumbersome because default value of alpha needs to be varied if same operation has to be performed on machine of different capabilities. User tuning is required for load balancing in case of heterogeneous data type and size.

(3) Dynamic load balancing through dynamic load task allocation works well and makes efficient utilization of GPUs available for all kinds of data. It does not require tuning and performs equally well making full utilization of GPUs even if machine changes (as it automatically tunes itself to available GPU capabilities).

(4) Dynamic load balancing provides ease of use, flexibility, efficient GPU utilization to user as it automatically scales the work to utilize all available resources.

Future work includes reformulation of the load balancing problem as a knapsack problem in combinatorial optimization, with an objective of allocation of tasks to each resource to minimize time subject to an upper limit.

6. ACKNOWLEDGMENT

This project was funded by the National Plan for Science, Technology and Innovation (MAARIFAH), King Abdulaziz City for Science and Technology, Kingdom of Saudi Arabia, Award Number (34/959).

7. REFERENCES

[1] Jespersen, D.C., 2010. Acceleration of a CFD code with a GPU. Scientific Programming, 18(3-4), pp.193-201.

[2] Xu, R., Tian, X., Chandrasekaran, S. and Chapman, B., 2015. Multi-GPU support on single node using directive-based programming model. Scientific Programming.

[3] Lee, J.H., Nigania, N., Kim, H., Patel, K. and Kim, H., 2015. OpenCL performance evaluation on modern multicore CPUs. Scientific Programming, 2015, p.4.

[4] J., Varbanescu, A.L. and Sips, H., 2011, September. A comprehensive performance comparison of CUDA and OpenCL. In Parallel Processing (ICPP), 2011 International Conference on (pp. 216-225). IEEE.

[5] Karimi, K., Dickson, N.G. and Hamze, F., 2010. A performance comparison of CUDA and OpenCL. arXiv preprint arXiv:1005.2581.

[6] Bradski, G. and Kaehler, A., 2008. Learning OpenCV: Computer vision with the OpenCV library. " O'Reilly Media, Inc.".

[7] OpenCV, GPU Module Introduction. [online] http://docs.opencv.org/modules/gpu/doc/introduction.html

[8] Sanders, J. and Kandrot, E., 2010. CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional.

[9] Kirk, D.B. and Wen-mei, W.H., 2010. Programming massively parallel processor. Morgan Kaufmann.

[10] Nielsen, I. and Janssen, C.L., 2008. Multicore challenges and benefits for high performance scientific computing. Scientific Programming, 16(4), pp.277-285.

[11] Lan, Z., Taylor, V.E. and Bryan, G., 2002. Dynamic load balancing of SAMR applications on distributed systems. Scientific Programming, 10(4), pp.319-328.

[12] Parent, J., Verbeeck, K., Lemeire, J., Nowe, A., Steenhaut, K. and Dirkx, E., 2004. Adaptive load balancing of parallel applications with multi-agent reinforcement learning on heterogeneous systems. Scientific Programming, 12(2), pp.71-79.

[13] OpenCV Test data. [online] Available at: https://github.com/itseez/opencv_extra.

[14] Caltech 256 database, J2K and 256_object category, http://www.csee.wvu.edu/~xinl/database.html

[15] Standard test Image, online http://www.imageprocessingplace.com/root_files_v3/image_databases.html

8. APPENDIX A: CODE SAMPLE

This appendix provides code sample of multiple GPU abstraction layer for bilateral filteration of images.

```
void UserFunction(thread_data_t *data)
{           // 'Filename' is provided by the "data->name" ,
everything else is user defined

char ss[20]="out";
char *Inputpath=new char[400];
*Inputpath=NULL;
char *Outputpath=new char[400];
*Outputpath=NULL;
cout<<  " Processing: "<< data->name << endl;
cout<<endl;

Mat src = imread(strcat(strcat(Inputpath, data->InputPath),
data->name) , CV_LOAD_IMAGE_COLOR);

if (!src.data)
{
 cout << "data is not read"<< endl;
  exit(1);
}
gpu::GpuMat d_src2(src);
gpu::GpuMat d_dst2;
gpu::bilateralFilter(d_src2,d_dst2,-1, 50,7);
Mat dst2(d_dst2);

imwrite(strcat( strcat(Outputpath, data->OutputPath),
strcat(ss, data->name)), dst2);

cout<<" End of Processing: "<<data->name<<endl;

cout<<endl;
};
```

9. APPENDIX B: HARDWARE

Specification	GPU device 0: "Tesla C2075"	GPU device 1: "Quadro 2000"
CUDA Capability Major/Minor version number	2.0	2.1
Total amount of global memory:	4096 MB (4294967295 bytes)	1024 MB (1073741824 bytes)
Multiprocessors:	(14) x (32) CUDA Cores	(4) x (48) CUDA Cores
GPU Clock rate:	1147 MHz (1.15 GHz)	1251 MHz (1.25 GHz)
Memory Clock rate:	1566 MHz	1304 MHz
Memory Bus Width	384-bit	128-bit
L2 Cache Size:	786432 bytes	262144 bytes
Max Texture Dimension Size (x, y, z)	1D=(65536), 2D=(65536,65535), 3D=(2048,2048, 2048)	1D=(65536), 2D=(65536,65535), 3D=(2048,2048, 2048)
Max Layered Texture Size (dim) x layers	1D=(16384) x 2048, 2D=(16384,16384) x 2048	1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory	65536 bytes	65536 bytes
Total amount of shared memory per block	49152 bytes	49152 bytes
Total number of registers available per block	32768	32768
Warp size	32	32
Maximum number of threads per multiprocessor	1536	1536
Maximum number of threads per block	1024	1024
Maximum sizes of each dimension of a block	1024x1024x64	1024x1024x64
Maximum sizes of each dimension of a grid	65535x65535 x65535	65535x65535 x65535
Maximum memory pitch	2147483647 bytes	2147483647 bytes
Texture alignment	512 bytes	512 bytes
Concurrent copy and kernel execution	Yes with 2 copy engine(s)	Yes with 1 copy engine(s)
Run time limit on kernels	No	Yes
Integrated GPU sharing Host Memory	No	No
Support host page-locked memory mapping	Yes	Yes
Alignment requirement for Surfaces	Yes	Yes
Device has ECC support	Enabled	Disabled
CUDA Device Driver Mode (TCC or WDDM)	TCC (Tesla Compute Cluster Driver	WDDM (Windows Display Driver Model