# Parallel Video Processing

## [Parallel Programming Final Project Report] *

Po-Han Chen †
National Chiao Tung University
j.lin013@gmail.com

Cheng Sun ‡
National Chiao Tung University
s2821d3721@gmail.com

Yu-Wen Pwu §
National Chiao Tung University
yuwen41200@gmail.com

## ABSTRACT

Video processing typically requires immense amounts of computational resources. Devices supporting up to 4K resolution are currently hitting the market by storm. A greater demand for computation has thus arisen. Therefore, we set out to explore ways to boost the efficiency of video processing through a series of experiments. These experiments were conducted with a wide assortment of tools(or APIs) on multiple platforms. We've managed to achieve what we consider maximum possible speedups on our test platforms.

## 1. INTRODUCTION

Initially, we were interested in whether parallelizing simple image processing tasks could deliver any speedup. A number of image processing tasks rely on matrix multiplications which take only constant number of computations per pixel. This gives O(N) complexity and likely higher memory access frequencies.

Seeing these computations are reasonably simple, we suspected that compilers might optimize programs to an extent where speeding up with parallelization might not be possible. To test this out, we designed and conducted 2 experiments, Lightup and Lightup2. The former does simple arimetic calculations on each pixel, while the latter multiplies each pixel by $1.125^{(x+y)/128}$. From these experiments, we concluded that increasing the amount of computations per thread does raise speedups and that different platforms(or OSes) along with different optimization options are not the reasons for the subtle speedups in Lightup.

Our research into parallel video processing branched out from here. We decided to have a closer look at the feasibility of using FPGAs to speedup video processing. We consider FPGAs a suitable choice for heterogeneous computing because FPGAs are essentially massively parallel processors. Fully customized hardware designs can lead to maximized optimization. In the end, we were able to obtain nearly 1000x speedup on a 1920*1080 dataset. This was also 100x faster than other approaches.

Unsatisfied with Lightup experiments, we've found and tested a tool called pmbw - Parallel Memory Bandwidth Benchmark / Measurement [1]. It's a tool written in assembly that tests important memory characteristics including maximum bandwidth on our test platforms. After examining the test results from pmbw, we believed a speedup of up to 3.2x should be attainable.

We then turned our attention to real-world problems. We decided to use White balance as our video processing application. The White balance algorithm [2] we used takes O(N) time and was simple enough for us to experiment with. We used 5 different parallelization approaches: Pthread_TDM, OpenMP_TDM, CUDA, CUDA_TDM and TaskParallel (We will detail these approaches in later contexts). While CUDA still delivers poor results due to frequent global memory accesses, Pthread and OpenMP managed to yield 3x speedups on the main test platform and 2x speedups on the secondary test platform. Both numbers were consistent with the test results from pmbw. We concluded that certain hardware characteristics do pose an "invisible ceiling" especially in I/O bound applications and CUDA on consumer-grade GPUs might not be suitable for this type of applications.

## 2. PROPOSED SOLUTIONS: LIGHTUP SERIES (CPU/GPU)

### 2.1 Lightup

For each pixel, $[R', G', B'] = [R, G, B * 2 + 5]$
Then normalize the RGB values back to [0, 255].
This would increase the overall blue contrast.

### 2.2 Lightup

For each pixel, $[R', G', B'] = [R, G, B * 1.125^{(x+y)/128}]$
Then normalize the RGB values back to [0, 255].
The blue channels would become mostly 0's (ie. image becomes yellowish) except for the bottom-right corner.

* We will discuss our parallelization methods with White balance altogether in later contexts.

## 3. PROPOSED SOLUTIONS: LIGHTUP2.5 (FPGA)

### 3.1 Lightup2.5

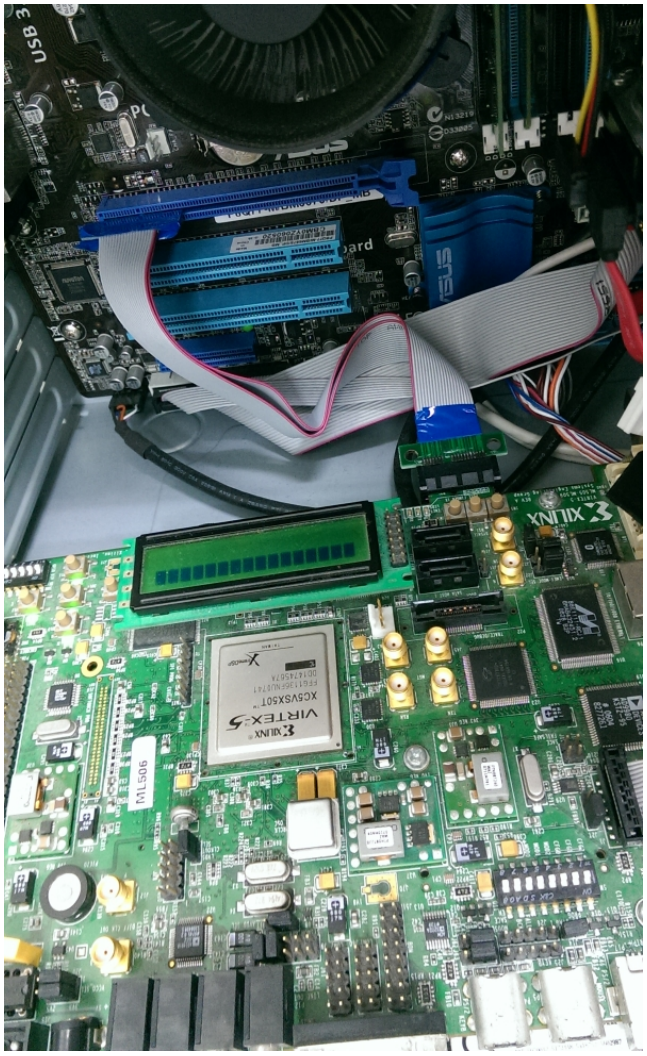
For each pixel, $[R', G', B'] = [R, G, B] * 1.125^{1024}$

## 3.2 Considerations

The performance of the circuit is highly dependent on the quality of the circuit design and the available resources on the FPGA development board. Although some EDA (electronic design automation) tools, like Xilinx SDAccel, support high-level synthesis that can convert OpenCL C/C++ codes into schematics, these techniques are not mature enough, and may lead to poor performance.

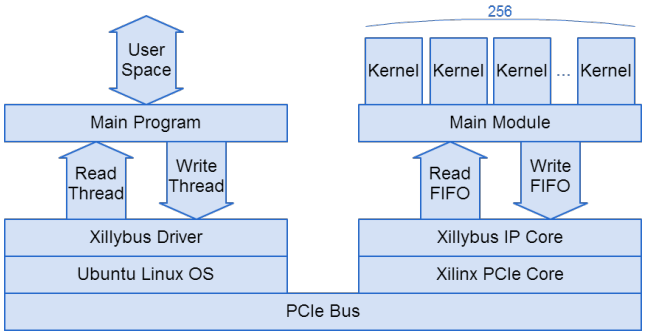
## 3.3 Implementation

We employ RTL (register-transfer level) design by the Verilog HDL (hardware description language), and we use the Xillybus IP core [3] for data transmission over the PCIe interface. Our targeted board is Xilinx Virtex-5 ML506 Evaluation Platform. We use C++ for software design. The host programs run on a 64-bit Linux.



## 3.4 Computation Platform

The PC-end (host) is on the left side, whereas the FPGA-end (device) is on the right side.




### 3.4.1 *Data Interface*

To hosts, the device looks just like a file. So we can use low-level (POSIX) file I/O functions to read/write data from/to the device, i.e. the FPGA board.

### 3.4.2 *Multithreading on Software*

For best performance, we have 2 threads implemented on the host program. The first is used to send data, the other is used to receive data.

## 3.5 Kernel Instantiation

We use a *generate block* to instantiate 256 kernel modules, namely, 256 simultaneously running threads. Each kernel module calculates a single color value, so the FPGA can calculate at most 256 values at the same time.

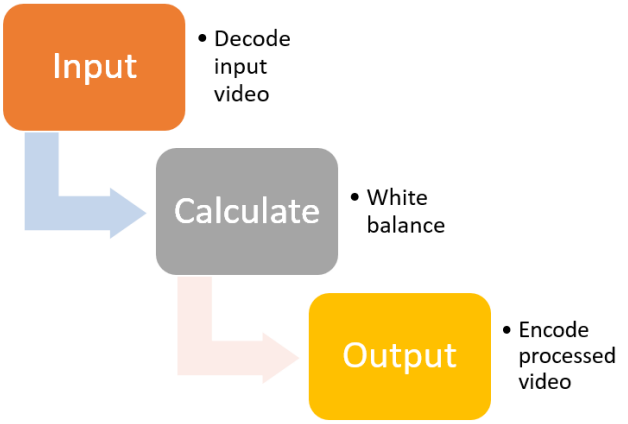
### 3.5.1 *Finite State Machine*

4 states: IDLE_STATE, RECV_STATE, EXEC_STATE, and SEND_STATE. After the program starts, the circuit goes to the second state. The second state waits and receives 256 color values, until then, it goes to the third state. The third state processes the 256 received values in parallel, then it goes to the last state. The last state waits and sends 256 new values back, and goes to the first state again.

# 4. PROPOSED SOLUTIONS: WHITE BALANCE

## 4.1 Block diagram

Input and Calculate are parallelizable.



## 4.2 Algorithm

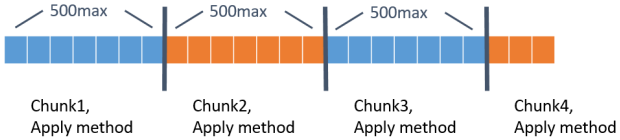
1. Calculate the averages of Red, Green and Blue values, as denoted by (AvgR, AvgG, AvgB)

2. Using Green as reference, for each pixel,
   $Red_{new} = Red * AvgG/AvgR$
   $Blue_{new} = Blue * AvgG/AvgB$

3. If $Red, Blue_{new} > 255$ Then $Red, Blue_{new} = 255$

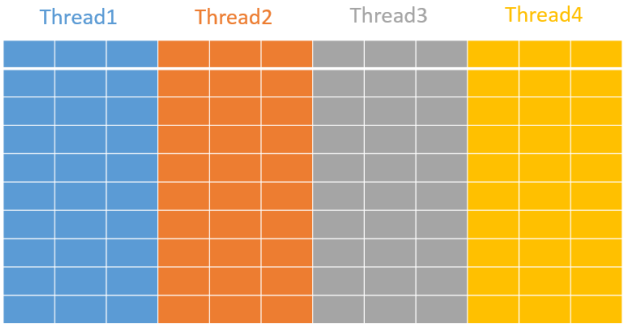
## 4.3 Initialization

With scalability in mind and to prevent memory exhaustion, we divide the input video into chunks with each containing no more than 500 frames.



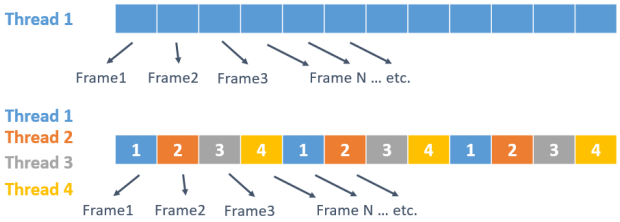

## 4.4 Parallel method1: Pthread

We process one frame at a time. We let each thread process a portion of columns.




## 4.5 Parallel method2: Pthread_TDM

Utilizing C++11 threading libraries, we divide the input video by time. Each thread is assigned a collection of video frames as indicated by the following figure.

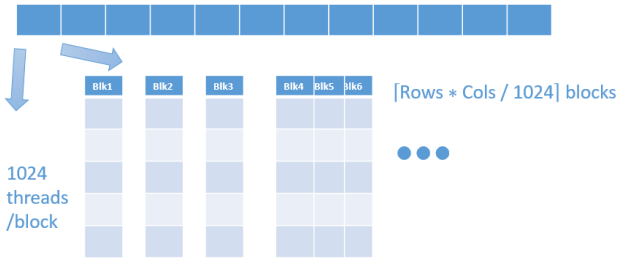



## 4.6 Parallel method3: OpenMP_TDM

We also divide the input video by time, but with OpenMP as the API.

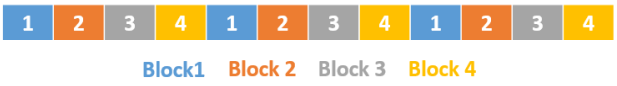
## 4.7 Parallel method4: CUDA

We process one frame in each iteration(as in memcpy operation pairs). Each thread processes one pixel.



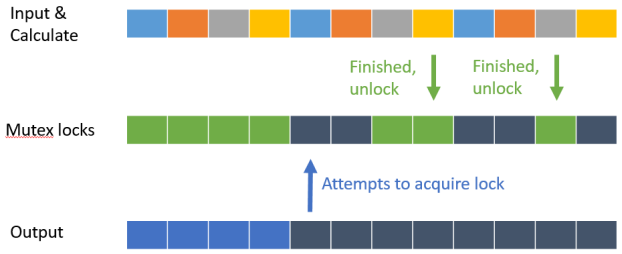

## 4.8 Parallel method5: CUDA_TDM

Very similar to previous approaches, we divide the input video by time where each thread block processes one frame.




## 4.9 Parallel method6: Pthread_with_TaskParallel

Extended from Pthread_TDM, rather than waiting for Input and Calculate to finish, we create a separate thread specifically for output.




# 5. EXPERIMENTAL METHODOLOGY

## 5.1 Environment specifications

APIs: OpenCV 2.4.11, OpenCV 3.0

| OS | CPU | RAM | Compiler |
|---|---|---|---|
| Windows 10 | i7-3770(4C8T) | 16GB | VS12, gcc 5.2.0 |
| Ubuntu 14.04 | i5-4200H(2C4T) | 8GB | gcc 4.8.4 |
| OSX El Capitan | i5-4260U(2C4T) | 8GB | gcc 5.2.0 |
| Ubuntu 15.10 | 2 vCores | 2GB | gcc 5.2.0 |

GPU (on i7 PC): GeForce GTX 670, CUDA v7.5, CUDA Capability 3.0.

FPGA Board: Xilinx Virtex-5 ML506 Evaluation Platform. FPGA Host PC: Ubuntu 14.04, i5-3570(4C4T), 8GB, gcc 5.3.0.

## 5.2 Input datasets

### *5.2.1 Lightup & Lightup2:*

360*240 / 1280*720 / 1920*1080, 148 frames (MP4).

### *5.2.2 Lightup2.5:*

256*256 / 1920*1080*3 (Raw data).

### *5.2.3 White balance:*

1280*720, 1422Frames (MP4, AVI).

## 5.3 Output

### *5.3.1 Lightup & Lightup2:*

During visual inspection, the programs display the output video in realtime.

### *5.3.2 Lightup2.5:*

Raw data (arrays) of sizes 256*256 / 1920*1080*3.

### *5.3.3 White balance:*

If output is opted in (through C macro definition), the programs produce videos in the same resolution as input in AVI format.

# 6. EXPERIMENTAL RESULTS

## 6.1 Lightup

Slowdown when input is small. Tiny speedup even when input is large. Similar results across OpenCV 2.4, Ubuntu, OSX.

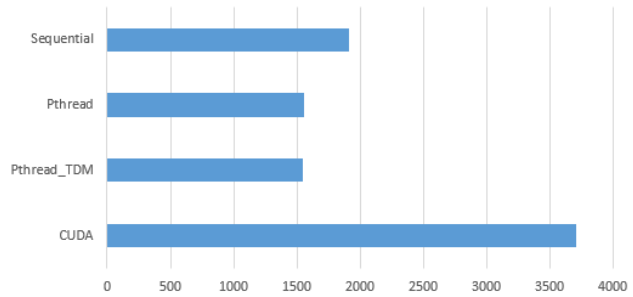



**Figure 1: Lightup (4 Threads, i7-3770, OpenCV3, VS12, Windows 10)**

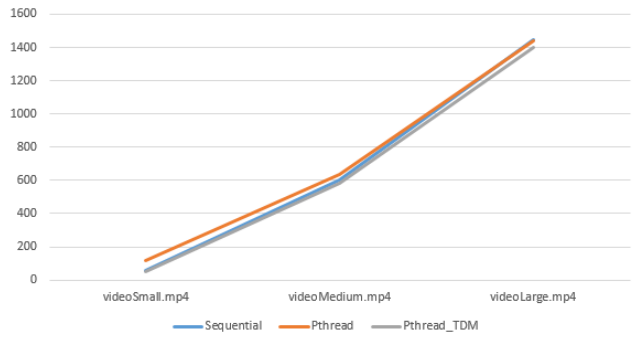



**Figure 2: Lightup (4 Threads, i7-3770, OpenCV3, Cygwin-gcc 5.2.0, Windows 10)**

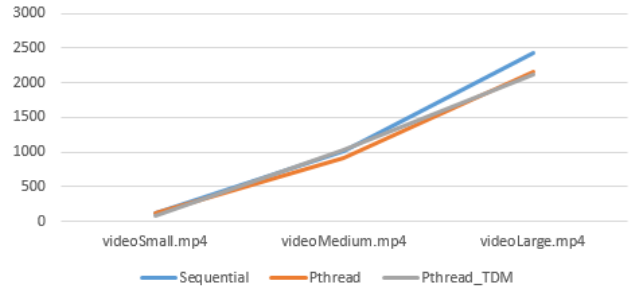



**Figure 3: Lightup (4 Threads, i5-4260U, OpenCV3, gcc 5.2.0, OSX El Capitan)**

## 6.2 Lightup2

The results are clearly better than Lightup. Pthread_TDM seems to be better than Pthread.

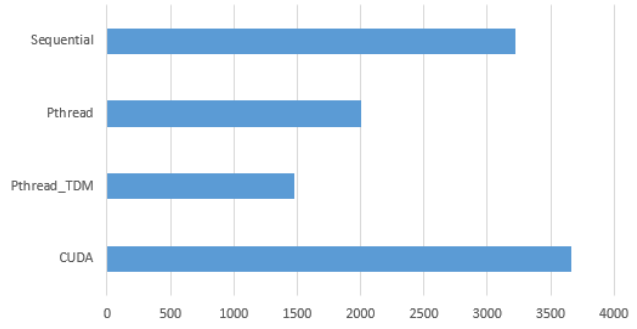



**Figure 4: Lightup2 (4 Threads, i7-3770, OpenCV3, VS12, Windows 10)**

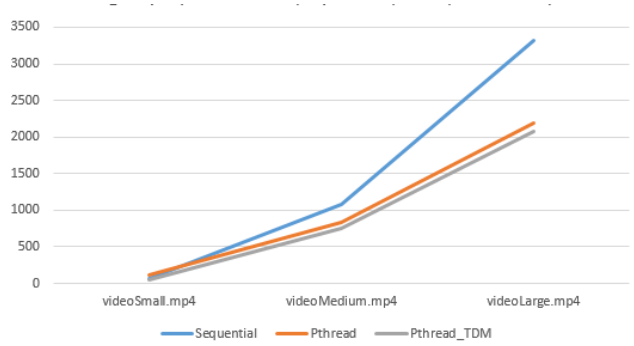



**Figure 5: Lightup2 (4 Threads, i7-3770, OpenCV3, Cygwin-gcc 5.2.0, Windows 10)**

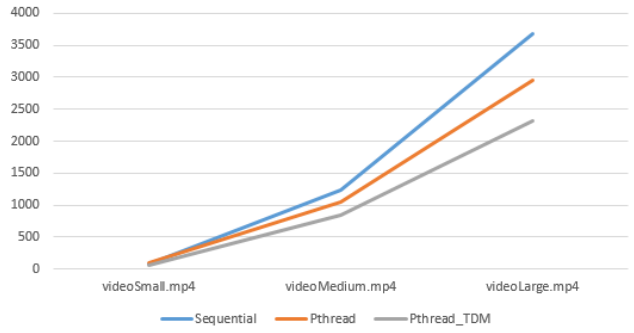



**Figure 6: Lightup2 (4 Threads, i5-4200H, OpenCV3, gcc 4.8.4, Ubuntu 14.04)**

## 6.3 Lightup2.5 (FPGA)

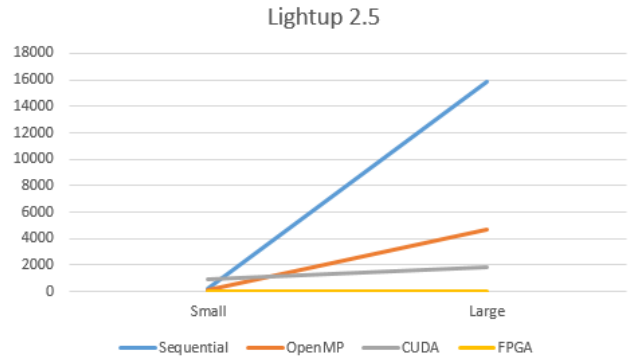
Whopping speedups. FPGA is at least 100x faster than other methods.

**Table 1: Lightup2.5 Runtimes**

| Size | Sequential | OpenMP | CUDA | FPGA |
|---|---|---|---|---|
| Small | 179.096ms | 67.667ms | 891.333ms | **0.534ms** |
| Large | 15873.200ms | 4678.667ms | 1799.333ms | **16.104ms** |

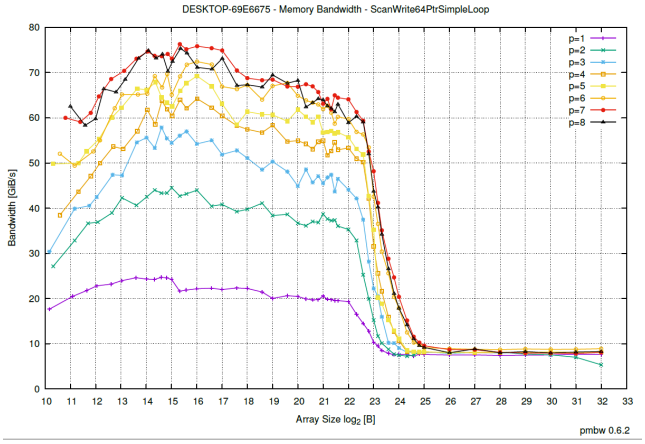
**Table 2: Lightup2.5 Speedups**

| Size | Sequential | OpenMP | CUDA | FPGA |
|---|---|---|---|---|
| Small | 1.00x | 2.65x | 0.20x | **335.39x** |
| Large | 1.00x | 3.39x | 8.82x | **985.67x** |




## 6.4 pmbw - Parallel Memory Bandwidth Benchmark / Measurement

From the results below, we conclude that I/O should be parallelization-worthy (ie. speeding up is possible). Also, the upper bound looks to be around 3.2x (on the main test platform).




## 6.5 White balance

**Figure 7: White balance Results: Before and After**

**2 Threads**, i7-3770, VS12, Windows 10:
Pthread_TDM: 1.69x speedup
OpenMP_TDM: 1.77x speedup
CUDA: 0.80x slowdown
CUDA_TDM: 0.41x slowdown

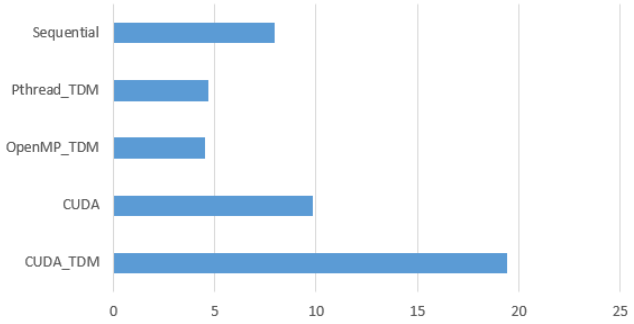



**Figure 8: 2 Threads, i7-3770, VS12, Windows 10**

## 6.6 White balance and more threads

We can see our approaches toward white balance do benefit from more threads. The highest speedup we get is consistent with the results from pmbw.

**Table 3: White balance Speedups with more threads**

| Threads | 1 | 2 | 4 | 6 | 8 |
|---|---|---|---|---|---|
| Pthread_TDM | 1.00x | 1.69x | 2.49x | 2.76x | 3.02x |
| OpenMP_TDM | 1.00x | 1.77x | 2.76x | 2.92x | 3.00x |

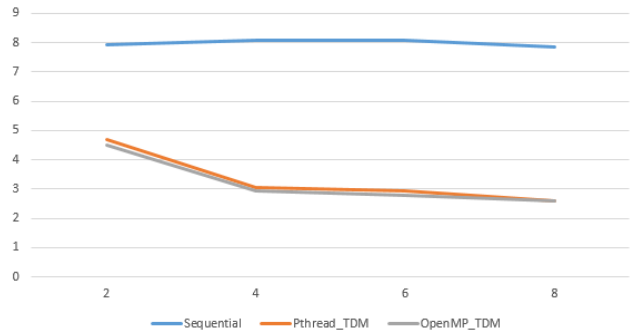



**Figure 9: White Balance (2 to 8 Threads, i7-3770, VS12, Windows 10)**

## 6.7 White balance and other platforms

Positive results across different OSes and compilers.

NB: 2 to 4 Threads, i5-4200H, gcc 4.8.4, Ubuntu 14.04.
VPS: 2 Threads, 2 vCores, gcc 5.2.0, Ubuntu 15.10.
Cygwin: 8 Threads, i7-3770, Cygwin-gcc 5.2.0, Windows 10.

**Table 4: White balance Speedups on other platforms**

| Platform | Seq | NB | VPS | Cygwin |
|---|---|---|---|---|
| Pthread_TDM | 1.00x | 2.00x | 1.36x | 3.13x |
| OpenMP_TDM | 1.00x | 1.76x | 1.27x | 3.20x |

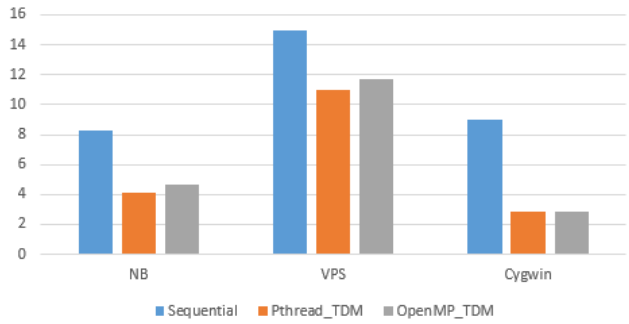



**Figure 10: White Balance (NB, VPS, Cygwin)**

## 6.8 White balance Profiling results

When output is taken into account, the whole process is output-bound.

| Method | Total | Input | Calculate | Output |
|---|---|---|---|---|
| Sequential | 100% | 29.41% | 11.59% | **59.00%** |
| OpenMP | 100% | 21.66% | 7.29% | **71.04%** |

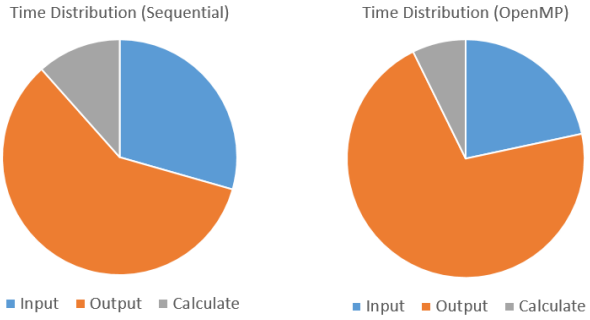



**Figure 11: Profiling results (NB)**

## 6.9 White balance when output is accounted

Pthread_with_TaskParallel delivers better results(ie. higher efficiency) when output is taken in account.

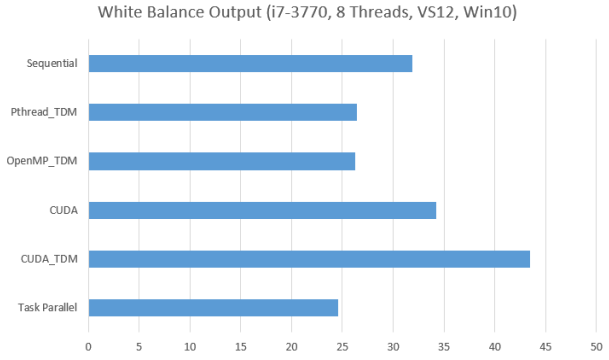



**Figure 12: White balance with task parallel on main PC**

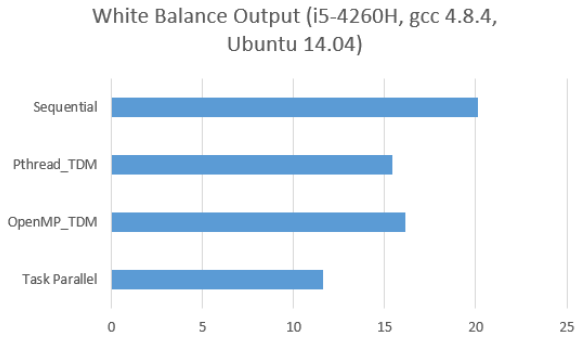



**Figure 13: White balance with task parallel on NB**

# 7. RELATED WORK


- Ching-Chih Weng, Homer Chen, and Chiou-Shann Fuh. A Novel Automatic White Balance Method For Digital Still Cameras. In *IEEE International Symposium on Circuits and Systems*, 2005.

- Dennis Lin, Xiaohuang (Victor) Huang, Quang Nguyen, Joshua Blackburn, Christopher Rodrigues, Thomas Huang, Minh N. Do, Sanjay J. Patel, and Wen-Mei W. Hwu. The Parallelization of Video Processing. In *IEEE Signal Processing Magazine*, pp. 103-112, November 2009.

- Mihalis Psarakis, Aggelos Pikrakis, Giannis Dendrinos. FPGA-based Acceleration for Tracking Audio Effects in Movies. In *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012.

- Matthew Jacobsen, Yoav Freund, Ryan Kastner. RIFFA: A Reusable Integration Framework for FPGA Accelerators. In *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012.


# 8. CONCLUSIONS

- Our experimental results show that certain hardware characteristics, memory bandwidth in particular, seem to be closely relevant to the efficiency of programs with linear complexities. We experienced a maximum 3.2x speedup on the Win10 desktop and a maximum 2x speedup on the Ubuntu laptop. Both were consistent with the pmbw test results from each of their platforms. It seems true that speedups do have upper bounds which depend on the hardware.

- Additionally, we think we should keep simple computations on the CPU end. Using NVIDIA's Nsight profiling tool, we were able to identify huge memory bottlenecks over the course of the program execution. It is perhaps futile to parallelize simple linear-complexity algorithms as the time it takes to copy the data between host and device would be relatively significant. Things could be better on a workstation-grade GPU though.

- Moreover, it seems that FPGAs handle parallel programming extremely well. Past research has shown that using FPGAs as accelerators can realize better performance than using GPGPUs. It does hold true in our experiments. In our opinion, it's probably safe to say that FPGAs are great tools for applications demanding massive computations.

- Lastly, the video codec used for input also seems to matter. When the input video is encoded in mp4 format, parallelizing the input procedure actually slows down the whole process. It took us quite a bit of effort to accomplish the results we have right now. A reasonable guess for this would be, perhaps non-sequential reads caused slight delays in between and longer decoding times potentially triggered unwanted queueing effects.

# 9. ENDNOTES


- We would like to thank Prof. Yi-Ping You, Assistant Professor at the Department of Computer Science, National Chiao Tung University and the teaching assistants for an excellent course (Parallel Programming, Fall 2015). The knowledge we gained throughout the course helped us tremendously.

- We would also like to thank Prof. Chun-Jen Tsai, Associate Professor at the Department of Computer Science, National Chiao Tung University for lending the FPGA board (ML506 Evaluation Platform) to us and for all the advices he offered during the research.


- Collaborated project. All sources are publicly available on GitHub:
`https://github.com/sunset1995/parallel_analysis`
`https://github.com/yuwen41200/fpga-computing`

# 10. REFERENCES


[1] Timo Bingmann. (2013). pmbw - Parallel Memory Bandwidth Benchmark / Measurement. Available at `http://panthema.net/2013/pmbw/`

[2] Jason Su. (2010). Illuminant Estimation: Gray World. Available at `http://web.stanford.edu/~sujason/ColorBalancing/grayworld.html`

[3] Xillybus Ltd. (2016). An FPGA IP Core for Easy DMA over PCIe with Windows and Linux [Online]. Available at `http://xillybus.com/.`