Single Thread, OpenMP and CUDA Benchmarking Based on Vanilla Computer Vision Pipeline

Aytekin Erdogan
Middle East Technical University/ Ericsson
MSc Student/ Software Developer
aytekin.erdogan@metu.edu.tr/ aytekin.erdogan@ericsson.com

***Abstract*—In this study, we have desinged a vanilla computer vision pipeline based on convolution, batch normalization, activation layer. We have coded the pipeline for cuda, single thread cpu and openmp frmo scratch. We measure the performance of each methods based on timing.**

***Index Terms*—cuda, openmp, benchmarking, computer vision**

I. Introduction

In the age of parallel computing and era of AI, a question comes to mind "Can all the processes be made parallel? If so, what is the performance improvement of parallel computing in terms of timing?" In this study we aim to answer these questions. We will be carrying out a similar study as in the case of [1]. In the context of this concern, a benchmark on parallel programming libraries and frameworks will be done based on a vanilla computer vision [1] pipeline.

II. Problem Definition

We would like to create a timing benchmark of cuda, openACC openMP/MPI and single thread implementation of a vanilla Convolutional Neural Network from scratch. In a supervised image classification/segmentation task within the context of machine learning, a model is trained on input-output pairs, where inputs are images and outputs are corresponding classes and/or images. For this project we will be implementing the basis of such systems. At first stages, we will be implementing the forward pass steps consisting of:

- Image Preprocessing
- Batch Normalization
- 2D Convolution layers,
- Activation (ReLU) layers,

For this purpose, several deep learning layers must be implemented by understanding their basics such as convolution, batch normalization and activation layers. Later, possible test setups must be written to report the output results and timing for these codes. Solution pipeline can be given as:

A. Preprocessing Step

Let X is a set of grayscale images whose dimensions are [B, W, H, 1]. Here, B indicates the batch size of this list. As expected, W and H denote the spatial resolution of images. Hence, we would like to utilize the processing power fully, we will operate on multiple images at the sametime (e.g., 32 images, thus B=32) and resize them into the same resolution (e.g., W=512 and H=512) by leveraging OpenCV library or different libraries. For simplicity we plan to work on single channel images at first stages. So, we will convert them into grayscale images for simplicity (i.e., channel size is 1). Later, they will be concatenated to form the batch list X whose dimensions are [B, W, H, 1].

[1]The code is available at: https://github.com/aytekine/ComputerVisionCuda

B. Convolution Layer

This layer takes a set of images X and a set of kernels W (i.e, [K, L, 1, C]) as input and computes a representation Y whose dimensions are [B, W, H, C]. This basically computes a convolution operation between images and kernels. For the parameter of the convolution kernel we will predefine kernels since we plan first to create an inference pipeline and then a learning network. For predefined kernels we plan to use W as [[1, 2, 1], [0, 0, 0], [-1, -2, -1]] and [[0.11, 0.11, 0.11], [0.11, 0.11, 0.11], [0.11, 0.11, 0.11]]. Note that individual kernels also will be concatenated and formed as [3, 3, 1, 2].

C. Batch Normalization Layer

This layer takes the output of convolution layer Y and estimates a normalized representation denoted as Z. Similarly, the dimensions of the output are [B,W, H, C]. Basically, mean and variance values are estimated by considering B, W and H channels. Note that the last channel (C) is not used in the calculation of mean and variance values. Hence, the mean and variance dimensions are C. Later; the input data Y is normalized with these values.

D. Activation Layer

Similarly, the output of batch normalization layer Z is fed to this layer and the output is denoted as V where the dimensions are [B, W, H, C]. Note that this layer is a pixel-wise operation and a simple function V=max(Z,0) can be utilized where the values less than zero are set to 0.

E. Visualization

The output of activation layer V must be visualized for each channel and image (i.e., [W, H, 1]).

III. Literature Review

Similar studies, regardless of computer vision tasks, are carried out in literature. For example in [1], the author compared the serial CPU, CUDA, OpenMP and MPI packages for a vanilla matrix summation task. For the comparison the results are shared as:

- OpenMP vs Serial CPU: 7.1x faster CudaSlow vs Serial CPU: 10.5x faster
- CudaFast vs Serial CPU: 82.8x faster
- CudaSlow vs OpenMP: 1.5x faster
- CudaFast vs OpenMP: 11.7x faster

where CudaFast and CudaSlow two different algorithms with different compute times.

Considering the cuda implementations for computer vision tasks, exploiting parallelism is a common strategy for accelerating convolution. Parallel processors keep getting faster, but algorithms such as image convolution remain memory bounded on parallel processors such as GPUs. Therefore, reducing memory communication is fundamental to accelerating image convolution. To reduce memory communication, in study [2], they reorganized the convolution algorithm to prefetch image regions to register, and claimed to do more work per thread with fewer threads. Depending on filter size, they claimed their speedups on two NVIDIA architectures range from 1.2× to 4.5× over state-of-the-art GPU libraries. As in the case of study [3], there are lots of benchmarking studies which tend to compare different implementations of the same algorithm on GPU. In [3], they propose a new implementation of convolution operation which limits the memory back and forth operations with adaptive tiling operation. Same algorithmic approach also applies for study [4] and [4]: in study [4] by leveraging spatially-varying kernels they also implement a convolution operation benchmark based on gpu. In study [5] they implement a gaussian filtering and give a performance benchmark.

IV. Limitations

In order to have a fair comparison, we should be able to implement the same vanilla network for all cuda, openacc, openmp and CPU code while fully utilizing the capabilities of each hardware and algorihm in terms of bandwidth and computation resources.

V. Solution Pipeline

As discussed earlier we will be implementing a simple computer vision project from scratch both on single thread cpu, multithread with openMP and CUDA. Algortihm block diagram can be seen at the Figure 1.Each and every solution consists of the following steps:

- Preprocessing (input: image directory output: X[B,W,H,1])
- Convolution Layer (input: X[B,W,H,1], W[K,L,1,C] output: Y[B,W,H,C])
- Batch Normalization (input: Y[B,W,H,C] output: Z[B,W,H,C])
- ActivationActivation Layer (input: Z[B,W,H,C] output: V[B,W,H,C])
- Visulization (input: V[B,W,H,C] output: each img[W,H,1])

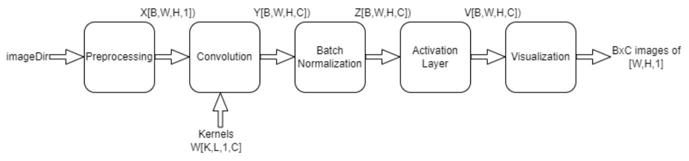


Fig. 1. Computer Vision Pipeline used for benchmarking.

VI. Experiment Details

In this part of the report we will be talking through the implementation of each algorithm on cpu, openMP and CUDA. For the sake of comparison instead of already available built-in library functions, we would like to implement each layer from scratch.

For instance, for the single thread cpu part: there is already available std::convolution and boost::convolution algorithms, which are capable of using multithread implementation implicitly. This situation may cause a problem on a fair comparison since we are trying to compare single thread cpu, multithread cpu and gpu.

a) Preprocessing Step: We will operate on multiple images at the same time (e.g., 32 images, thus batch size (B) is 32) and resize them into the same resolution (e.g., W=512 and H=512). Output is X with dimensions [B, W, H, C].

b) Convolution Layer: This layer takes a batch of images X and a set of kernels W (i,e, [K, L, 1, C]) as input and computes a representation Y with dimensions [B, W, H, C]. This basically computes a convolution operation.

c) Batch Normalization Layer: This layer takes the output of convolution layer Y and estimates a normalized representation Z using mean and variance across batch and spatial dimensions.

d) Activation Layer: Similarly, the output of batch normalization layer is fed to this layer. Output dimensions are [B, W, H, C]. This layer is a pixel-wise operation and a simple function V=max(Z,0).

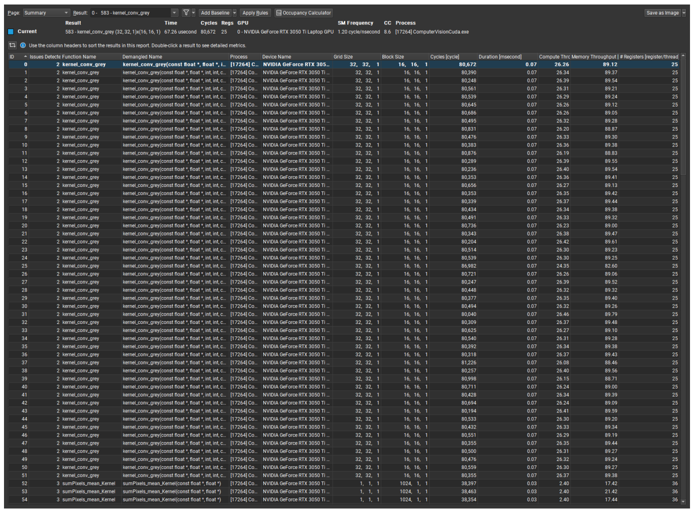
e) Visualization: The output of activation layer must be visualized for each channel and image (i.e., [W, H, 1]).

VII. Results

For the hardware, we have complited the tests on the following hardware and test configuration:

- CPU: Intel i7 10710UJ CPU @1.1 GHz 12 CPU
- GPU: nvidia RTX3050 Ti Laptop GPU
- Batch size: 4
- Total Image Number: 28
- Input Image Size: 1908 x 1200

As promised, a vanilla computer vision pipeline is tested and timed on different hardware selection. Timing results can be seen at Table I. By the help of Table I we have generated the Table II which shows the speed up when compared to the each other. For the cuda part, used grid size, block size, memory usage and any relavent details are given at the Figure 2.


Page: Summary | Result: 0 - 583 - kernel_conv_grey | Add Baseline | Apply Rules | Occupancy Calculator | Save as Image

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	583 - kernel_conv_grey (32, 32, 1)x(16, 16, 1)	67.26 usecond	80,672	25	0 - NVIDIA GeForce RTX 3050 Ti Laptop GPU	1.20 cycle/nsecond	8.6	[17264] ComputerVisionCuda.exe

Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.

ID	Issues Detecte	Function Name	Demangled Name	Process	Device Name	Grid Size	Block Size	Cycles [cycle]	Duration [msecond]	Compute Thro	Memory Throughput	# Registers [register/thread
0	**2**	**kernel_conv_grey**	**kernel_conv_grey(const float *, float *, i...**	**[17264] C...**	**NVIDIA GeForce RTX 305...**	**32, 32, 1**	**16, 16, 1**	**80,672**	**0.07**	**26.26**	**89.12**	**25**
1	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,390	0.07	26.34	89.37	25
2	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,248	0.07	26.39	89.54	25
3	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,561	0.07	26.31	89.21	25
4	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,539	0.07	26.29	89.24	25
5	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,645	0.07	26.26	89.12	25
6	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,686	0.07	26.26	89.05	25
7	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,495	0.07	26.32	89.28	25
8	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,831	0.07	26.20	88.87	25
9	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,476	0.07	26.33	89.30	25
10	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,383	0.07	26.36	89.38	25
11	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,876	0.07	26.19	88.83	25
12	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,289	0.07	26.39	89.55	25
13	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,236	0.07	26.40	89.54	25
14	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,353	0.07	26.36	89.41	25
15	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,656	0.07	26.27	89.13	25
16	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,353	0.07	26.35	89.42	25
17	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,339	0.07	26.37	89.44	25
18	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,434	0.07	26.34	89.38	25
19	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,491	0.07	26.33	89.32	25
20	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,736	0.07	26.23	89.00	25
21	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,343	0.07	26.38	89.47	25
22	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,204	0.07	26.42	89.61	25
23	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,514	0.07	26.30	89.23	25
24	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,539	0.07	26.30	89.25	25
25	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	86,982	0.07	24.35	82.60	25
26	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,721	0.07	26.26	89.06	25
27	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,247	0.07	26.39	89.52	25
28	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,448	0.07	26.32	89.32	25
29	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,377	0.07	26.35	89.40	25
30	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,494	0.07	26.32	89.26	25
31	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,040	0.07	26.46	89.79	25
32	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,309	0.07	26.37	89.48	25
33	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,625	0.07	26.27	89.10	25
34	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,540	0.07	26.31	89.28	25
35	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,392	0.07	26.34	89.38	25
36	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,318	0.07	26.37	89.43	25
37	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	81,226	0.07	26.08	88.46	25
38	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,257	0.07	26.40	89.56	25
39	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,998	0.07	26.15	88.71	25
40	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,711	0.07	26.24	89.00	25
41	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,428	0.07	26.34	89.39	25
42	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,694	0.07	26.24	89.09	25
43	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,194	0.07	26.41	89.59	25
44	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,533	0.07	26.30	89.20	25
45	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,432	0.07	26.33	89.34	25
46	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,551	0.07	26.29	89.19	25
47	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,355	0.07	26.35	89.44	25
48	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,500	0.07	26.31	89.27	25
49	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,476	0.07	26.32	89.24	25
50	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,559	0.07	26.30	89.27	25
51	2	kernel_conv_grey	kernel_conv_grey(const float *, float *, int, int, c...	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	32, 32, 1	16, 16, 1	80,355	0.07	26.37	89.38	25
52	3	sumPixels_mean_Kernel	sumPixels_mean_Kernel(const float *, float *)	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	1, 1, 1	1024, 1, 1	38,397	0.03	2.40	17.42	36
53	3	sumPixels_mean_Kernel	sumPixels_mean_Kernel(const float *, float *)	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	1, 1, 1	1024, 1, 1	38,463	0.03	2.40	21.42	36
54	3	sumPixels_mean_Kernel	sumPixels_mean_Kernel(const float *, float *)	[17264] Co...	NVIDIA GeForce RTX 3050 Ti ...	1, 1, 1	1024, 1, 1	38,354	0.03	2.40	17.44	36

Fig. 2. Nsight Profiler

According to table, all images are seperated by grids, while convolution kernels, mean and variance calculations, activation layers are seperated by blocks.

TABLE I
METHODS' TIMINGS

Computer Vision Modules	**HW Occupation Times in ms**		
	Single Thread CPU	***OpenMP(12 thread)***	***CUDA***
Preprocess	35.6	20.6	NA
Convolutions	1401.1	299.7	13
Batch Normalization	422.9	131.7	63
ReLU	1054.8	1001.8	1
Overall Timing	2914.4	1453.1	68

TABLE II
METHODS' SPEED-UP

Computer Vision Modules	**Speed-Ups compared to each other**		
	CPU to OpenMP	***CPU to CUDA***	***OpenMP to CUDA***
Preprocess	1.72	NA	NA
Convolutions	4.67	107.77	23.1
Batch Normalization	3.21	6.71	2.1
ReLU	1.1	1054.8	1001.8
Overall Timing	2.0	42.85	21.4

A. Nsight Profiler Suggestions

It is also wise to state that those implemantations are not optimized and can be optimized as follows:

- Some of kernels are utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit.
- Some kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical (100.0%) and measured achieved occupancy (86.8%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.
- Some kernel grids are too small to fill the available resources on this device, resulting in only 0.1 full waves across all SMs.
- Some kernel's theoretical occupancy (66.7%) is limited by the required amount of shared memory which is limited by the number of required registers.

VIII. CONCLUSION

We have successfully completed the experiments based on the pre-defined computer vision pipeline. In the mean time, we have implemented all pipeline from scratch. Due to fact that, algorithms are not optimized based on the hardware configuration. The reason behind this choice is: not to test how well the CUDA's optimized X algorithm outperform the single thread cpu's optimized Y algorithm. Our aim is to test compare the timing of a set of operation on different computing method. As further studies, cuda code can be optimized as suggested by the Nsight code profiler.

REFERENCES

[1] https://forums.developer.nvidia.com/t/when-to-use-serial-cpu-cuda-openmp-and-mpi/48379

[2] F. N. Iandola, D. Sheffield, M. J. Anderson, P. M. Phothilimthana and K. Keutzer, "Communication-minimizing 2D convolution in GPU registers," 2013 IEEE International Conference on Image Processing, 2013, pp. 2116-2120, doi: 10.1109/ICIP.2013.6738436.

[3] B Werkhoven,"Optimizing convolution operations on GPUs using adaptive tiling" 2018 IEEE International Conference on Image Processing, 2018, pp. 1916-1920.

[4] Hartung, S., Shukla, H., Miller, J. P., Pennypacker, C. (2012, September). GPU acceleration of image convolution using spatially-varying kernel. In 2012 19th IEEE International Conference on Image Processing (pp.1685-1688). IEEE.

[5] K. Preethi and K. S. Vishvaksenan, "Gaussian Filtering Implementation and Performance Analysis on GPU," 2018 International Conference on Inventive Research in Computing Applications (ICIRCA), 2018, pp. 936-939, doi: 10.1109/ICIRCA.2018.8597299.