[ICMTEST]
IC™ Value: 3.00

ISSN: 2277-9655
Impact Factor: 4.116
CODEN: IJESS7

IJESRT
INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

GPGPU (GENERAL PURPOSE COMPUTING USING GRAPHICS PROCESSING UNIT) ON BILATERAL FILTER

Prof. Ashish A. Shastrakar[*1] **& Prof. Prasad Sangare2**
[*1&2]Assistant Professor(ETC Dept.), G.H.R.I.E.T., Nagpur

ABSTRACT

Production chains featuring industrial vision are becoming more and more widespread. Those processes are often heavy and applied to high definition images with important frame rate. Powerful calculators are thus needed to follow the ever growing production rate.

NVIDIA is currently designing interfaces providing a CUDA (Compute Unified Device Architecture) architecture allowing parallel data computation. This could increase the performance of every operating system using graphical processing units (GPU). Pure Data, thanks to its graphical modular development environment, allows fast prototype developments.

Those factors led us to start a research program dedicated to the realizations of image processing modules for Pure Data written in CUDA. So in this we have implemented image blurring algorithm to get familiar with GPGPU

KEYWORDS: Image Processing, CUDA, Algorithm, Optimization, GPGPU (General Purpose Computing using Graphics Processing Unit)

I. INTRODUCTION

The objective of this project is to develop and implement CUDA-written modules in a visual development environment. The purpose is to accelerate the already existing image processing modules by dispatching a part of the process to the GPU.

This application can be used also for industrial, medical or artistic purposes. Image Blurring is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination.

II. CUDA

The CUDA multi-core architecture allows parallel data computing. It can improve performances by using the NVIDIA graphics cards. CUDA requires a driver that uses a streaming technique for communication between the GPU, called *device*, and the CPU (Central Processing Unit), called the host. Every program written in CUDA follows four important steps.

First, data to be processed by the GPU are copied in its memory. Then the CPU transmits the instructions that will be executed on the graphics card . Once these previous steps have been accomplished, the results of those operations are copied in a local memory.

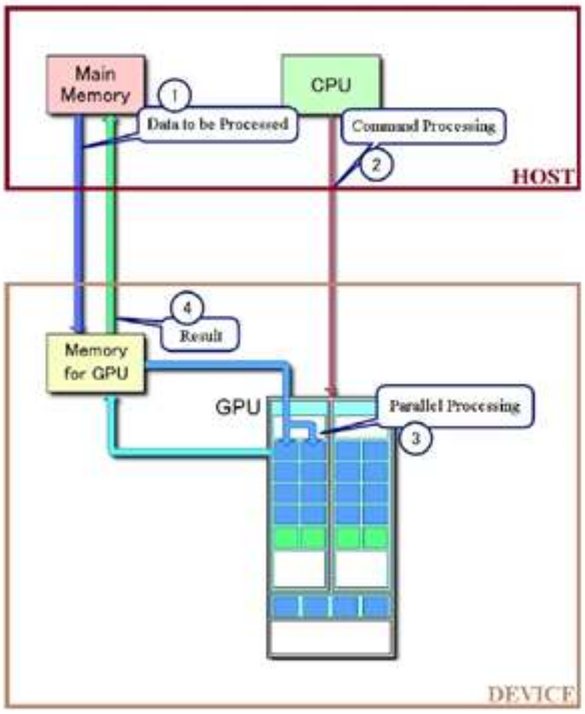

Figure 1: Programming steps in CUDA

In order to obtain good performances, massively parallelizable phases must be executed on the *device.*
In image processing, the CPU and GPU have a distinct role to play:

CPU: Reading the picture
GPU: Processing
CPU: Action to be done depending on processing to be do
CPU: Memory cleaning

III. ARCHITECTURE

A GPU which counts a strict minimum of 32 cores (average is 128) is called the grid.

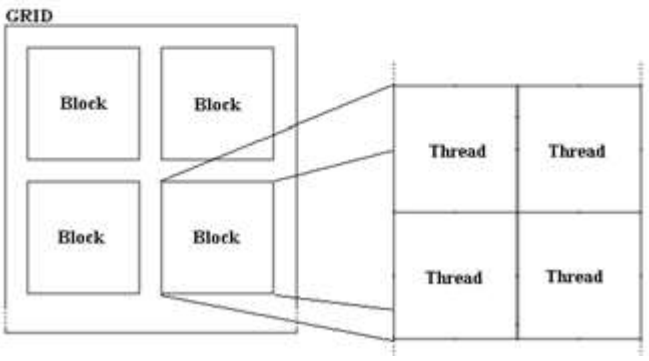

Figure 2: Grid subdivision in threads

This grid is divided into a given number of *blocks*. This number is defined according to the task to be undertaken at programming time. Each of those *blocks* contains multiple *threads* once the code has been written and taking into account the hardware (GPU cores number). A *thread* is the smallest subdivision of a task.

Blocks are independent. In other words they don't depend on the results coming from other blocks. Threads running inside a particular block can thus only communicate with threads from this block

IV. IMPLEMENTATION

As already said, a CUDA program consists of two parts : one part is running on the *host* and the other is running on the *device*. The complete code can fit in one file (with the ".cu" extension), describing those two parts.

CUDA extends the C language by bringing 9 new key words, 24 new types and 62 new functions.

CUDA software architecture, suitable for running the CUDA program, consists of three parts: a driver responsible for transmitting calculation from application to GPU.

A runtime offering an interface between the GPU and the application. A bundle of libraries.

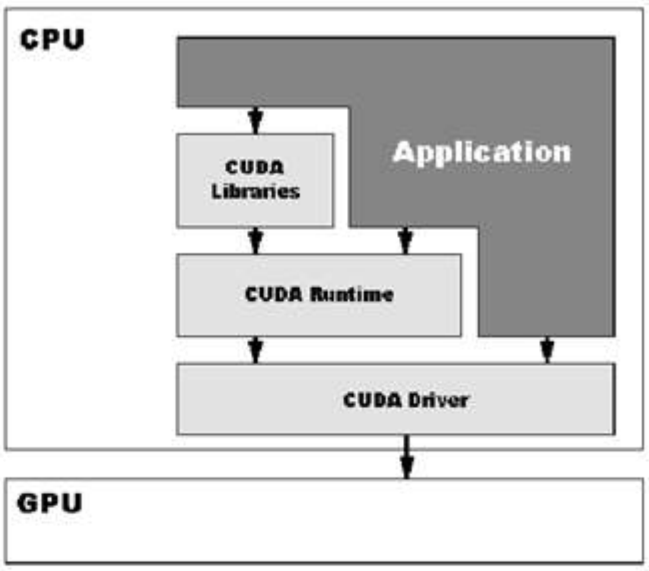

Figure 3: Software architecture of CUDA

The heart of CUDA is its *kernels. A kernel* is a portion of code to be executed in parallel on the *device*. Each instance of a *kernel* is itself a *thread*.

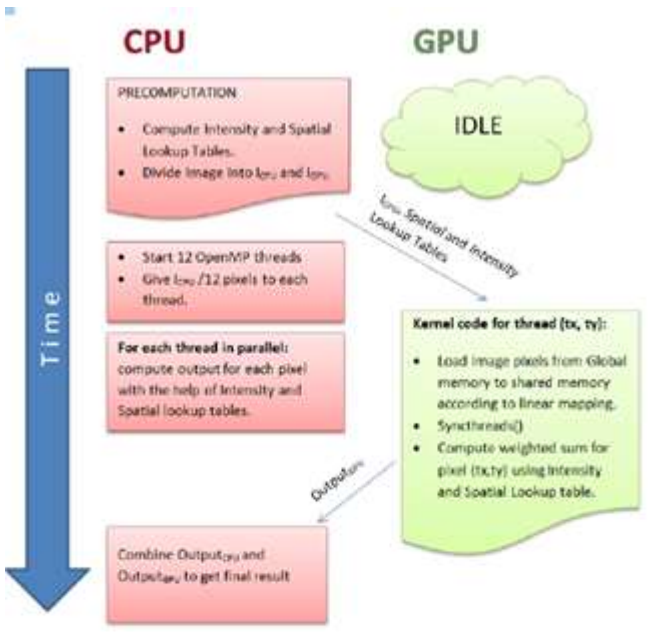
CPU
GPU
Time
PRECOMPUTATION
Compute Intensity and Spatial Lookup Tables.
Divide Image into I_{CPU} and I_{GPU}
IDLE
I_{GPU}, Spatial and Intensity Lookup Tables
Start 12 OpenMP threads
Give I_{CPU}/12 pixels to each thread.
For each thread in parallel: compute output for each pixel with the help of Intensity and Spatial lookup tables.
Kernel code for thread (tx, ty):
Load image pixels from Global memory to shared memory according to linear mapping.
Syncthreads()
Compute weighted sum for pixel (tx,ty) using Intensity and Spatial Lookup table.
$Output_{GPU}$
Combine $Output_{CPU}$ and $Output_{GPU}$ to get final result

Figure 4: Flow chart of algorithm with time

In our hybrid CPU+GPU algorithm for bilateral filtering, The pixels can be divided among threads, where each thread computes the weighted sum for that pixel. The spatial filter can be computed once we know the filter

V. IMAGE BLURRING

For Blurring purpose the image taken is of an animated Dino of size 512*512. On this image the mask of 9*9 will be applied for blurring it. I take an existing image blurring algorithm, developed by Mario Klingemann, to demonstrate the way to do image blurring with CUDA. In this sample, there are some minor code changes with CUDA for this algorithm and we see how CUDA can speed up the performance.

Blur image which is always a time consuming task. Blurring quality and processing speed cannot always have good performance for both. CUDA might help programmers resolve this issue. This code is tested on Windows 7 with NVIDIA GT 610.

Blur needs to process image rows first and then columns. There are two while loops to process rows and columns of image consecutively. The time consuming parts are the outer while loops for image rows and columns.

Therefore, they are the targets to be modified by CUDA. For normal image processing following are the important parameters required for blurring of image:

unsigned long* **pImage** [in/out]: 32-bit image buffer
unsigned **w** [in]: Image width unsigned **h** [in]: Image height
unsigned **r** [in]: Blur level

On implementing the blurring in a normal conventional way ,the processing time is 0.063553 (ms), tested by CPU. We will then see how performance can be speed up by CUDA

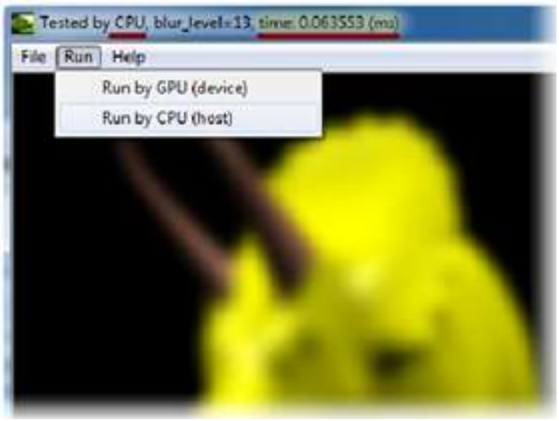

Figure 5: Burring on CPU

Now let us see the code with CUDA.The most significant parts are stack buffers which need to have independent buffers for each row and column. Because threads are running in parallel, stack buffers have to be separated and used by individual rows and columns. The rest of the code has nothing much changed except for the CUDA codes.

Now to speed up the algorithm using CUDA, we had make some changes in the parameters which are as follows:
uchar4* pImage [in/out]: 32-bit image buffer
uchar4* stack_data_horiz_ptr [in]: Stack buffer for rows
uchar4* stack_data_vert_ptr [in]: Stack buffer for columns
unsigned w [in]: Image width unsigned h [in]: Image height
unsigned r [in]: Blur level bool bMapped [in]: Flag of support of "host memory mapping to device memory"

As a result ,we see that the processing time is only 0.000150 (ms), tested by GPU. The processing time with CUDA is 300x or more faster than the conventional way

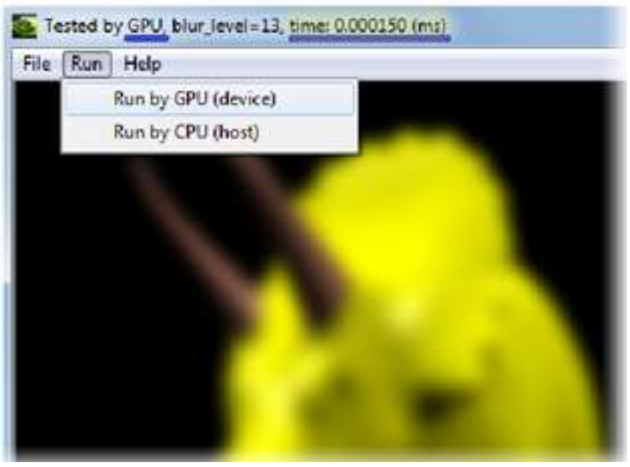
Tested by GPU, blur_level=13, time: 0.000150 (ms)

Figure 6: Blurring on GPU

VI. RESULT AND CONCLUSIONS

The result apparently tells that parallel computing with CUDA is amazing. It was noted that in order to get advantage of the GPU, several processes are to be processed and/or the calculations are to be complicated enough.

VII. REFERENCES

[1] J. Sanders and E. Kandrot: CUDA by exemple - An Introduction to General-Purpose GPU Programming, Addison-Wesley, Michigan, October 2010.
[2] D. B. Kirk and W. W. Hwu: Programming Massively Parallel Processors - A Hands-on Approach, Morgan Kaufman, 2009.
[3] J. Kreidler: Jloadbang - Programming Electronic Music in Pure Data,Wolke Verlag, Hofheim, 2009.
[4] [4] NVIDIA. CUDA ZONE. http://www.nvidia.com/cuda.
[5] General Purpose GPU Programming (GPGPU) Website, http://www.gpgpu.org.