Imperial Journal of Interdisciplinary Research (IJIR)
Vol-3, Issue-4, 2017
ISSN: 2454-1362, http://www.onlinejournal.in

Cuda Image Processing for Multicore Processor

Ankita Banginwar [1], Geeta Deokar[2], Pranoti Bisen[3] & Sakshi Gupta[4]
Department Of Computer Science and Engineering, Sant Shri Gajanan Maharaj College of Engineering Shegaon, SGBAU, India.

Abstract: *Traditional methods for processing large images are extremely time intensive. Also, conventional image processing methods do not take advantage of available computing resources such as multicore central processing unit (CPU) and manycore general purpose graphics processing unit (GP-GPU). By applying parallel programming techniques to various image filters should improve the overall performance without compromising the existing resources. Parallel implementation of image processing on Compute Unified Device Architecture (CUDA)-accelerated CPU/GPU system has potential to process the image very fast. In this algorithm, CUDA-accelerated image processing method is used for multicore/manycore systems.*
NVIDIA CUDA is an API for C, C++, and FORTRAN programming languages, which has been developed by NVIDIA for parallel programming on GP-GPUs. This enables programmers to use the NVIDIA graphics card for massively parallel programming. Graphics Processing Unit (GPU) is a processor on a graphics card specialized for compute intensive, highly parallel computation.

Introduction

Image processing is a well known field. It is a form of signals processing in which the input is an image, and the output can be an image or anything else that undergoes some meaningful processing. Altering an image to be brighter, or darker is an example of a common image processing tool that is available in basic image editors. Often, processing happens on the entire image, and the same steps are applied to every pixel of the image. This means a lot of repetition of the same work. Newer technology allows better quality images to be taken.

A simple high-level image processing takes an input image as a matrix and passes it through another matrix such as convolution matrix. The output/resultant image is the convoluted image. The convolution matrix can also be called a filter. Image processing and filtering application has traditionally been a time and resource consuming task given the presence of more and more pixels as the file/image size increases. A sequential image processing approach has been replaced by a parallel programming approach usually done through a CPU as the number of available cores has increased. However, the speedup factor is restricted to the number of cores present in the CPU. The appearance of the multithreaded parallel programming capabilities provided by CUDA opened a way to increase the speedup factor by hundreds to thousands, restricted by the number of cores in the GP-GPU.NVIDIA CUDA architecture provides an efficient means to access massively-parallel threaded GPUs to achieve a higher degree of performance and energy efficiency. Since image processing works on pixels as a matrix of elements, each thread can work independently on each pixel, providing a higher computational speed, performance, and energy efficiency through a flexible application programming interface (API) and programming model. Hence the available CUDA technology, help to improve the image processing and filtering performance. In this work, we develop a CPU/GPU based image processing technique using CUDA/C programming to process large images very fast.

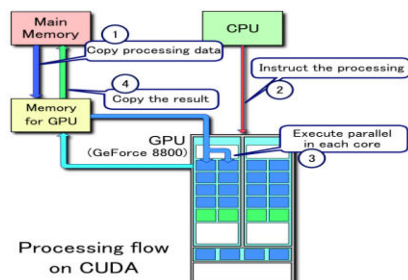
Image processing is a well-known field. It is a form of signals processing in which the input is an image, and the output can be an image or anything else that undergoes some meaningful processing. Altering an image to be brighter, or darker is an example of a common image processing tool that is available in basic image editors. Often, processing happens on the entire image, and the same steps are applied to every pixel of the image. This means a lot of repetition of the same work. Newer technology allows better quality images to be taken.

A simple high-level image processing takes an input image as a matrix and passes it through another matrix such as convolution matrix. The output/resultant image is the convoluted image. The convolution matrix can also be called a filter. Image processing and filtering application has traditionally been a time and resource consuming task given the presence of more and more pixels as the file/image size increases. A sequential image processing approach has been replaced by a parallel programming approach usually done through a CPU as the number of available cores has increased. However, the speedup factor is restricted to the number of cores present in the CPU. The appearance of the multithreaded parallel programming

capabilities provided by CUDA opened a way to increase the speedup factor by hundreds to thousands, restricted by the number of cores in the GP-GPU.NVIDIA CUDA architecture provides an efficient means to access massively-parallel threaded GPUs to achieve a higher degree of performance and energy efficiency. Since image processing works on pixels as a matrix of elements, each thread can work independently on each pixel, providing a higher computational speed, performance, and energy efficiency through a flexible application programming interface (API) and programming model. Hence the available CUDA technology, help to improve the image processing and filtering performance. In this work, we develop a CPU/GPU based image processing technique using CUDA/C programming to process large images very fast.

2. NVIDIA CUDA

NVIDIA CUDA is an API for C, C++, and FORTRAN programming languages, which has been developed by NVIDIA for parallel programming on GP-GPUs. This enables programmers to use the NVIDIA graphics card for massively parallel programming. Graphics Processing Unit (GPU) is a processor on a graphics card specialized for compute intensive, highly parallel computation.. It has millions of transistors, much more than the Central Processing Unit. A GPU card has many cores which are smaller than the ones present on the CPU but can execute many tasks in parallel. Since GPUs have the ability of executing a huge number of jobs in parallel through CUDA, the programmer is able to execute general purpose computation on GPU cards [8, 9]. And, CUDA has kernel functions which can be called from the host/CPU and will be executed on the device/GPU. The kernel will be executed by each thread simultaneously. In order to call the kernel, one needs to set the number of threads in each block and the number of blocks required for the calculations.



2.1. CUDA Programing Model

2.1. CUDA Programing Model

CUDA programming is a type of heterogeneous programming that involves running code on two different platforms: a host and a device. The host system consists primarily of the CPU, main memory and its supporting architecture. The device is generally the video card consisting of a CUDA-enabled GPU and its supporting architecture. The source code for a CUDA program consists of both the host and device code mixed in the same file. Because the source code targets two different processing architectures, additional steps are required in the compilation process. The NVidia C Compiler (NVCC) first parses the source code and creates two separate files: one to be executed by the host and one for the device. The host file is compiled with a standard C/C++ compiler which produces standard CPU object files. The device file is compiled with the CUDA C Compiler (CUDACC) which produces CUDA object files. These object files are in an assembly language known as Parallel Thread execution or PTX files.

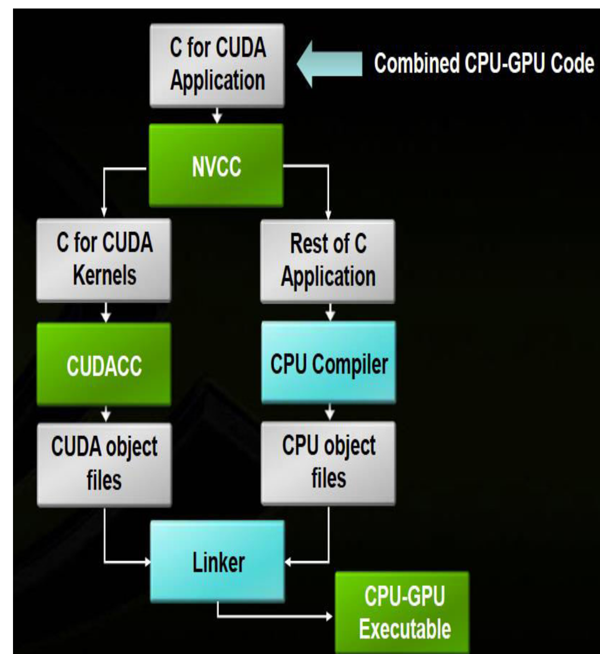


Figure 2.1.1 The compilation process for source file with host & device code

A CUDA program starts execution on the host. When it encounters the Kernel, it will launch the kernel and continues execution on the CPU without waiting for the completion of the kernel. The groups of threads created as a result of the kernel invocation is collectively referred to as a grid. The grid terminates when the kernel terminates. Currently in CUDA, only one kernel can be executed at a time. If the host encounters another kernel while a previous kernel is not yet complete, the CPU will stall until the kernel is complete. The next-generation architecture FERMI allows for the concurrent execution of multiple kernels.

2.2 CUDA Thread Hierarchy

Threads on the device are automatically invoked when a kernel is being executed. The programmer determines the number of threads that best suits the given problem. The thread count along with the thread configurations are passed into the kernel. The entire collection of threads responsible for an execution of the kernel is called a grid. A grid is further partitioned and can consist of one or more thread blocks. A block is an array of concurrent threads that execute the same thread program and can cooperate in achieving a result. All blocks must contain the same number of threads and thread structure. Each block can have a maximum of up to 512 threads. blockIdx and threadIdx provides the current block and thread index information.

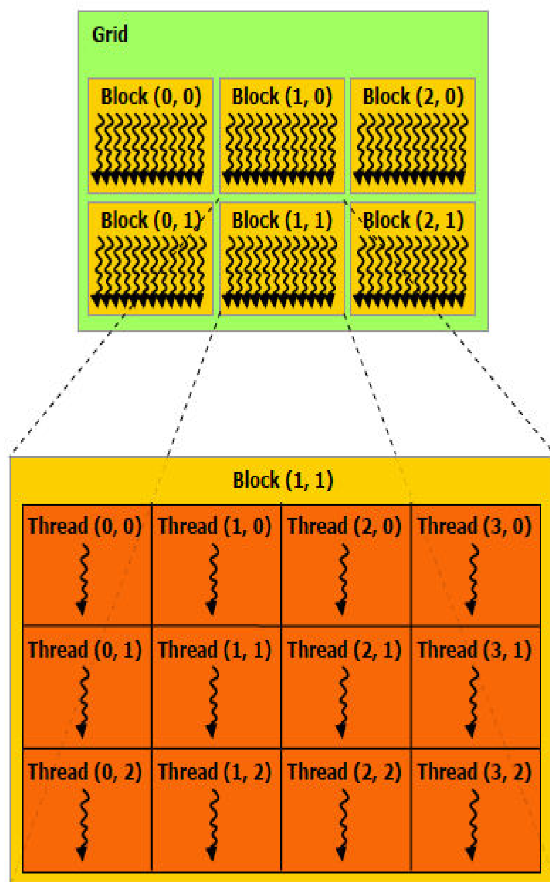


Figure 2.2.1 Grid of thread blocks

Once a kernel is launched, the corresponding grid and block structure is created. The blocks are then assigned to a SM by the SMC. Each SM executes up to 8 blocks concurrently. Remaining blocks are queued up until a SM is free. As shown in fig. 2.2.2 the more SM a graphics card has, the more concurrent blocks can be executed.

2.3 CUDA Memory

The typical ow of a CUDA program starts by loading data into host memory and from there transfer to device memory. When an instruction is executed, the threads can retrieve the data needed from device memory. Memory access however can be slow and have limited bandwidth. With thousands of threads making memory calls, this potentially can be a bottle neck and thus, rendering the SMs idled. To ease traffic congestion, CUDA provides several types of memory constructs that improve execution efficiency as shown in Fig. 2.3.1.

There are 4 major types of device memories: global, constant, shared and register memory.

- Global memory has the highest access latency among the three. A global variable is declared by using the keyword device. It is the easiest to use and requires very little strategy. It can easily be read and written to by the host using CUDA APIs and it can be easily accessed by the device. Global memory is the only way for threads from different blocks to communicate with each other.

- Constant memory is very similar to global memory. In fact, these are the only two memory that the host can read and write to. The main difference from global memory is that constant memory is read-only to the device because it is designed for faster parallel data access. A constant variable is declared by using the keyword constant. Like global memory, constant memory also lasts for the entire duration of the application.

- Shared memory is an on-chip memory that the host cannot access. This type of memory is allocated on a block level and can only be accessed by threads of that block. Shared memory is the most efficient way for threads of the same block to cooperate, usually by synchronizing read and write phases. It is much faster than using global memory for information sharing within a block. Shared memory is declared by using the keyword shared. It is typically used inside the kernel. The contents of the memory last for the entire duration of the kernel invocation.

- The last type of memory is register memory. Registers are allocated to each individual thread, and are private to each thread. If there are 1 million threads declaring a variable, 1 million versions will be created and stored in their registers. Once the kernel invocation is complete, that memory is released. Variables declared inside a kernel (that are not arrays, and without a keyword) are automatically stored in registers.

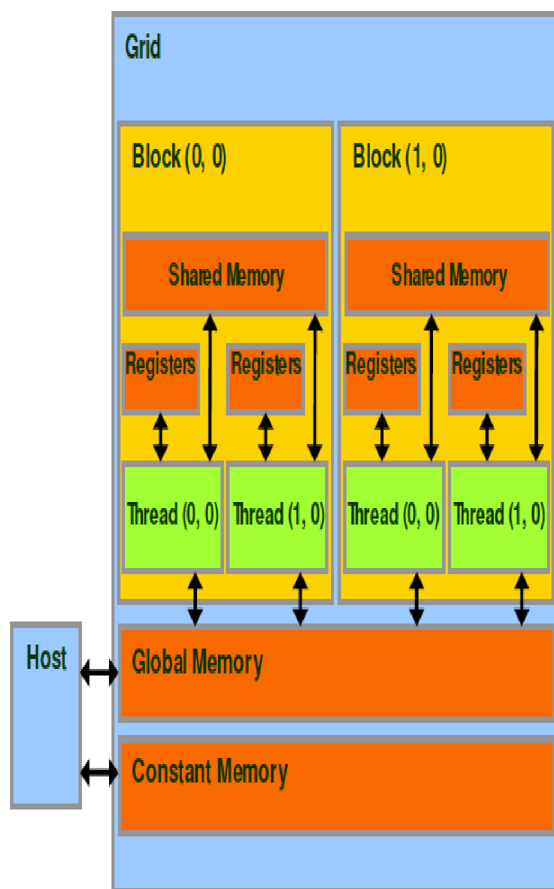


Figure 2.3.1 Different memory types: Constant, Global, Shared and Register memory

3. CUDA Accelerated Image Processing

In this, the sequential and parallel image processing implemented through C and CUDA/C programs. By processing, refer to the application of filters that change the image color properties, such as grayscale, invert, and others. In this section, there is briefly description of image processing algorithm for multicore/manycore systems, characteristics of the input image files, GPU parameters, and some important measurement parameters.

3.1. Image Processing Algorithm for Multicore/Manycore Computing

The image processing algorithm that is implemented in a parallel CUDA/C program. For traditional C and CUDA/C programs, the same image files are used as inputs. The algorithm considers the following general steps:

Image file related steps:

i) Get the input bitmap file name to be processed.
ii) Get the output file name to be generated.
a) This file contains the original pixels modified after applying the invert filter.
iii) Open the image file in read mode.
a) Parse the bitmap file header to get its properties including the pixels matrix width and height and the number of colours.
b) Update the file pointer to skip the header bytes.
c) Start a timer on the CPU.
Load the pixel's bytes in a one dimensional array with length equal to matrix width * matrix height * 3 (3 bytes per pixel)
d) Stop the timer started on step 3.
iv) Create the output file with the provided name.
v) Save the bitmap file header.

CUDA related steps:

vi) Allocate memory on host and device as needed. Then
a) Raise a CUDA Event (timestamp start).
- Allocate an array of integers with the same size as the one in step iii)c.

b) Raise a CUDA Event (timestamp end).
c) Raise a CUDA Event (timestamp start).
- Copy the loaded pixels into the space reserved on step vi)a.

d) Raise a CUDA Event (timestamp end).
vii) Raise a CUDA Event (timestamp start).
a) Kernel call with parameters including the number of blocks and threads.
b) Call thread synchronize from the CPU as a POSIX threads join equivalent .
viii) Raise a CUDA Event (timestamp end).
ix) Raise a CUDA Event (timestamp start).
a) Copy the modified pixels (pixels with applied filter) back to the CPU.
x) Raise a CUDA Event (timestamp end).
xi) Store the pixels in the output header.
xii) Free allocated memory on the device and the host

3.2. Input Image File

In this method, we use different bitmap input image files. Among the image files, size of a pixel-matrix is the only changing characteristic, where each pixel is defined by three bytes, i.e. a 24-bit color bitmap pictures. The smallest image used as input is 512x512 pixels, the next one 1,024x1,024 pixels and so on. We test, sequential and parallel programs with six input images. Due to the nature of bitmap files, no compression is considered so that the total amount of pixels in the pixel-matrix is directly accessed and used to perform the image filters calculation. Moreover, the absence of compression aims an important aspect to be evaluated: the total image size, ranging from 700 KB to 780 MB, to

measure and plot how the GPU handles the overhead caused by this dimension.

3.3. GPU Parameters

We use a CUDA server with an NVIDIA Tesla C2075 card that has the key features as shown in Table 3.3.1. Given the nature of the experiment, the features shown in the table are required in order to recreate and compare the experimental results due to the C and CUDA/C programs. It should be noted that the host PC has an Intel Xeon E5506 CPU with 8 cores.

CUDA/GPU Specification	Value / Description
Number of Streaming Processors (SMs)	14
Processor Cores per SM	32
Total Number of Cores	448
Capable of Running double precision	515 GFLOPs per sec
Memory	6 GB of GDDR5

TABLE 3.3.1. GPU key parameters

3.4. Measurement Parameters

Two important parameters/metrics for this study are speedup factor and CUDA overhead.

- **Speedup**: The ratio of the sequential execution time over the parallel execution time as shown in Equation 1.

  Speedup = Sequential time / Parallel time (1)

We measure speedup factors, for the actual pixels matrix manipulation (no CPU-to-GPU and GPU-to-CPU overhead-time due to copying data), where the invert filter is applied and the other one for the total execution time including the pixels reading time.

- **CUDA overhead**: The sum of the time required to allocate memory on device/GPU and the time taken to copy the initial values to the device and the time taken to copy the results from the device (to the host/CPU).

The different timestamps allow to plot and determine how much of the total time is destined for what CUDA operation for the smaller and larger images. It is important to note that the CUDA server may be shared by different user groups. An important assumption considers that the delay introduced by the shared nature of the CUDA enabled server affects both the sequential and parallel program execution times since the server is being used by other groups.

4. Result

As already mentioned in algorithm, it has implemented different CUDA Event timers to calculate how much time it takes for each section of the code to execute. With these timers, it show what kind of speedup we achieve for the actual work (i.e., manipulating the matrix on GPU) and for the total work (i.e., copying data from CPU to GPU, manipulating the matrix on GPU, and copying the result back to the CPU). The speedup due to the matrix manipulation (the actual work) on the GPU is illustrated in Figure 4.2. The speedup of the 512x512 pixels image is approximately 25x and the speedup of the 16,384x16,384 pixels image is approximately 365x.

There are two important observations:

He speedup increases with the increase of the matrix size (from 512x512 to 8,192x8,192).

(ii) we notice that the speedup factor does not grow as it does initially after a specific image size. The image with 8,192x8,192 pixels shows roughly the same (or slightly better) speedup factor as the image with 16,384x16,384 pixels. This is caused by the fact that each thread is working on several pixels rather than working on one or two pixels as in the smaller pictures since we use a maximum of 65,535 blocks and 512 threads in each block. Therefore, we have 33,553,920 threads which can work on different pixels. For the 8,192x8,192 pixels file, each thread is working just on 2 pixels. However for the 16,384x16,384 pixels image, each thread is working on 8 pixels.

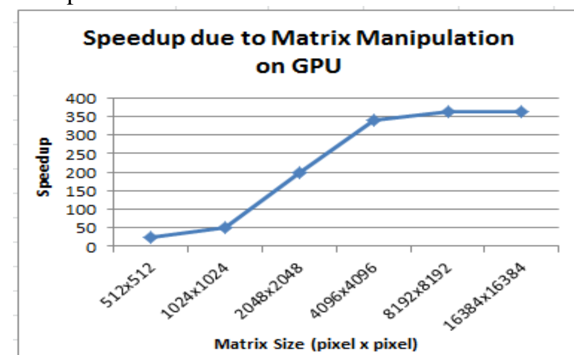


Fig. 4.1 Speedup factor due to the pixel-matrix manipulation on GPU cards

The impact of the image size on the overhead is depicted in Figure 4.2. It is observed that the CUDA malloc() time is approximately constant despite the image size. Hence, CUDA malloc() will be the main overhead for small images. It is observed that when the image size grows, the total execution time also grows because of the overhead time elapsed while copying the initial values from the CPU to the GPU and the results (i.e. modified pixels) from the GPU to the CPU. But still the overall parallel processing time for large size image is less as compare to serial processing time.

4. References:

[1] Abu Asaduzzaman, Angel Martinez, Aras Sepehri, "Time efficient image processing algorithm for multicore/manycore parallel computing," SoutheastCon, 2015.

[2] J. Tse, "Image processing with CUDA," University of Nevada,UNLV Theses/Dissertations Papers/Capstones, 2012.

[3] E. Young and F. Jargstorff, "Image processing and video algorithms with cuda," 2008.

[4] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on CUDA," International Conference on Computer Science and Software Engineering, Vol. 3,, pp. 198–201, 2008.

[5] N. Zhang, Y. shan Chen, and J. L. Wang, "Image parallel processing based on GPU," 2nd International Conference on Advanced Computer Sontrol (ICACC), Vol. 3, pp. 367–370, 2010.

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