**Contents**

**Abstract**

**Chapter1:Introduction to High performance computing**

**1.1 Need for High performance computing**

**1.2 Heterogeneous parallel computing**

**1.2.1 Parallel computing**

**1.2.2 Types of Parallelism**

**1.2.3 Classifications of Computer architecture**

**1.3. Heterogeneous computing**

**1.3.1 Heterogeneous architecture**

**1.3.2 NVIDIA GPU architecture**

**Chapter 2 : CUDA programming model**

**2.1 CUDA development cycle**

**2.1.1 Assess**

**2.1.2 Parallelize**

**2.1.3 Optimize**

**2.1.4 Deploy**

**2.2 CUDA Memory Hierarchy**

**2.2 Memory Management in CUDA**

**2.3 Organizing Threads in CUDA**

**2.5 CUDA programming structure**

### Chapter 3: Porting of Image processing and Computer vision algorithms

**3.1Adaptive contrast Enhancement**

**3.1.1 Algorithm Design**

**3.1.2 GPGPU Implementation**

**3.1.3 Results and Discussions**

**3.2 Motion De-blur algorithm**

**3.2.2 Algorithm Design**

**3.2.2.1 Estimating blur kernel**

**3.2.2.2 Deconvolution**

**3.2.3 GPGPU Implementation**

**3.2.4 Results and Discussions**

**Chapter 4: Porting of Autonomous drive algorithms**

**4.1 Embedded automotive platforms**

**4.1.1 Rcar-H3 SiP**

**4.1.2 Tegra X1 SoC**

**4.2 Temporal denoising**

**4.2.1 Algorithm Design**

**4.2.2 Multi-core CPU Implementation**

**4.2.3 Results and Discussion**

**4.3 Fog Rectification with denoising**

**4.3.1 Algorithm Design**

**4.3.2 Multi-core CPU Implementation**

**4.3.3 Results and discussions**

**4.4 Stereo disparity**

**4.4.1 Algorithm Design**

**4.4.2 Parallel GPU Streams based Implementation**

**4.4.3 Results and discussions**

**Chapter 5 : Conclusions and future scope**

**References**

**Appendix A**

**Abbreviations**

HPC - High Performance Computing

GPU - Graphics Processing Unit

ILP - Instruction Level Parallelism

CUDA - Compute Unified Device Architecture

GPGPU - General Programming on Graphics Processing Unit

SAD - Sum of Absolute Differences

AGCS - Automated Ground Control System

ARM - Advanced RISC Machine

RISC - Reduced Instruction Set Computer

NVVP - NVDIA Visual Profiler

PC - Personal Computer

CPU - Central Processing Unit

API - Application Interface

FFT - Fast Fourier Transform

IFFT - Inverse Fast Fourier Transform

SM - Streaming Multi-processor

SBC - Single Board Computer

PLT - Piece-wise Linear Transform

SiP - System in package

RAM - Random Access memory

SoC - System on Chip

MPcore - Multi Processor core

FPS - Frames Per Second

**Abstract**

A long standing challenge in the areas of computer vision, image processing, and autonomous drive is to achieve high accuracy and real time performance. The real time performance in autonomous drive area makes to deploy its applications into autonomous car, navigation systems, etc. The computer vision and image processing when using in the field of defense and military demands for real time processing. One of the most prominent situations is to process the video data received at ground station from Unmanned Aerial Vehicle(UAV). The ground station will have a specific hardware and the data received should be processed in real time. The instantaneous results makes the authority at the ground station to take an immediate action on any intrusions in the area under surveillance.

The demand to process the data in real time makes everybody to implement the algorithms effectively on to the target hardware. This is called the high performance implementations. The computer vision and image processing algorithms Adaptive contrast enhancement and motion de-blur are implemented on to a High performance system enabled with NVIDIA GPU. For the autonomous drive area an Embedded hardware powered by Rcar-H3 SiP is used. The autonomous algorithms temporal denoising, fog rectification are implemented to exploit the embedded hardware. As a special case to exploit the GPU based system, the stereo disparity algorithm is implemented on a system with NVIDIA GeForce GTX 1070 GPU.

The data parallelism techniques are described for Adaptive contrast enhancement, motion deblur and stereo disparity algorithms and these algorithms are implemented on GPU enabled systems. For high performance implementation on to Embedded hardware, techniques like pipelining, divide and process are explained. The pipelining concept is used for fog rectification where as divide and process methodology is implemented for temporal denoising.

**Chapter1: High-performance computing**

With the advancement in technology, In recent times, Artificial intelligence(AI) has evolved ,and has changed the way we produce,manufacture and deliver.Many terms have been formulated like autonomous systems, Cognitive computing, natural language processing and machine learning.And there is also a more advancement in the medical imaging and scientific computing domains .The common part in all these areas is that the processing data is huge, so needs more processing power than normal systems.

Processing of large amount of data takes significantly more time with the conventional hardware.For example, In autonomous systems to have a quicker response, the processing time should be less.And the scientific computing deals with simulations,analyses and predictions, so in this area getting quicker results saves time for the users.There are many such areas to list where quick processing of data is needed.One of the solutions to process the data quickly is increasing the processor frequency. But, the semiconductor industry is saturated with increasing the frequency of the processor and is reached a limit, beyond which the quantum effects which alter the operation of the semiconductor devices.

Due to the limits in increasing frequency of operation, we have moved on to multiple processing units to process the data in parallel. This processing of large data using many computing units is coined as High-performance computing, it includes designing algorithms, methodologies, and implementations to process data efficiently.In a single line, High-performance computing is the use of multiple processors or computers for running compute advanced applications efficiently, reliably and quickly with high throughput.Over the past few years high-performance computing has evolved considerably because of the emergence of CPU-GPU heterogeneous architectures.

**1.1 Need for High-performance computing**

Modern CPU's designed such that power available was plentiful and complete performance was the important figure of merit. The resulting architectures highly optimized for single-thread performance with the features such as out-of-order execution, branch prediction, and large primary instruction and data caches.In such type of designs, most of the power is used for data movement overheads, supplying the instructions and control mechanisms [5].

Data movement is the central dominating area for power dissipation.The data accessing time for on-chip static RAM(cache) is less than(~200 times) the time accessing DRAM.Current programming practice mainly focuses on sequential, homogeneous machines with a flat view of memory, present day systems (machines) moved away from this model.Increasing time and energy costs of accessing the distant memory need deeper memory hierarchies.With explicit control over the data movement within this hierarchies will result in performance sensitive code practices.Conventional architectures has a flat view of memory by having implicit data caching at multiple levels in the machine, such a practice is not sustainable for scalable systems.

The increased latency consequences in the machine will be hidden by adapting to increased amounts of parallelism, especially data and fine-grained thread parallelism.Conventional programming systems don't provide a convenient means of proving tens of, tens of thousands of threads (or even more) on a single die.A system with different kinds of processors which are complement to each other in performance comprises a heterogeneous system.In a heterogeneous system, individual processors will have processing elements with different performance characteristics, different views of the memory hierarchy, and will have levels of physical parallelism.

Programmers should select a particular core for processing their application, and the programming environment also should be able to determine which core is better for a given task.High-Performance computing provides all these methodologies and strategies, so the High-performance computing becomes future hope for super fast computing.

**1.2 Heterogeneous parallel computing**

Heterogeneous parallel computing is a hybrid word derived from two paradigms,heterogeneous computing and parallel computing.Heterogeneous computing is processing data using a heterogeneous architecture.Parallel computing is using multiple elements to process the data simultaneously.Heterogeneous parallel computing is processing the data in parallel using heterogeneous architecture.

**1.2.1 Parallel computing**

The main goal of parallel computing is to increase the speed of computation.Parallel computing is a form of computing, in which many computations are performed simultaneously, motivated by the fact that a large problem can be broken down into smaller ones, and all of them are solved concurrently.From the programmer’s point of view, it is mapping those parallel computations to the available compute resources.

There are two aspects to consider in parallel computing; those are hardware and software which are closely related to each other.The hardware aspect is dealing with the supportive computer architecture for parallel computing.Software aspect focuses on parallel programming by fully using the computational power of the computer architecture.The parallel execution of software is possible, such that the hardware must provide a suitable platform that supports execution of multiple threads.In the early age of semiconductor industry microprocessors are designed with a single core (unicore processor).Nowadays processors are fabricated with a prime motivation that they should be capable of doing parallel computing, they are made with more than one core(multi-core).The parallel programming can be viewed as a process that maps the computation of a problem to the available cores.

For sequential programming doesn’t require knowledge of the hardware architecture.But, implementing algorithms in parallel computing requires the knowledge of underlying computer architecture, in particular on the multi-core architectures.

**1.2.2 Types of Parallelism**

Parallel programming nowadays is important in computing world. The parallel execution of an algorithm gives good utilisation of hardware and the application executes faster.There exist mainly two types of parallelism in an application, task level parallelism and data level parallelism.The algorithm which is to be implemented using parallel computing should be analysed properly for the type of parallelism it supports.

Task-level parallelism is applicable if there exist many independent tasks or functions in an algorithm.For example, if there are three tasks in an application such as decoding video, processing video, and displaying the processed output.Even though each task has dependency with each other, they can be made independent by executing them using a three stage pipeline at the core level.We have used this methodology in implementing computer vision algorithms on a Rcar-H3 SoC which has an active quad-core ARM A-57.

Data parallelism, on the other hand, possible when there exist many data elements that can be operated independently.It focuses mainly on distributing data elements to multiple threads on multiple cores.This Data level parallelism is most useful in heterogeneous programming where the hardware contains a complementary coprocessor which is capable of handling the execution of multiple data elements at once.The algorithms implemented using HPC in this work are based on this parallelism. We followed CUDA(Compute Unified Device Architecture) programming to address the data parallel portions in the algorithms.

**1.2.3 Classifications of Computer architecture**

According to Flynn’s Taxonomy [6] there exist four types of computer architectures,

1. Single Instruction Single Data (SISD)

Traditional computer architecture is a serial architecture.They use unicore processors, at a time only one instruction stream is executed.

1. Single Instruction Multiple Data (SIMD)

These type of computers use multi core processors, all the cores execute the same instruction stream but operates on different data streams.This architecture is used extensively in this work.

1. Multiple Instruction Single Data (MISD)

This architecture is very rarely used where separate instruction streams will operate on same data streams.

1. Multiple Instruction Multiple Data (MIMD)

In this architecture different core will execute different instruction streams on different data.

Recently the computer architecture is moving from muti-core to many-core, many-core means having a high number of cores(tens or hundreds).For example a GPU represents a many core architecture, and it supports every type of parallelism we have described previously such as SI MD, MIMD,multi-threading and Instruction level parallelism(ILP).This type of architectures are called Single instruction multiple threading(SIMT) .Recently GPUs have been evolved to be more powerful so that they tackle massively parallel computations , and they are fully programmable for general purpose computing.

**1.3. Heterogeneous computing**

Heterogeneous computing refers to systems that use multiple types of processors. These are multi-core systems that gain performance not only by adding cores but also by incorporating special processing capabilities to specific tasks. These systems use multiple types of processors (typically CPUs and GPUs), usually on the same silicon chip or added discretely to the central processor.GPU processing, aside from the well-known 3D graphics rendering, it can be used for Mathematically intensive calculations on vast amounts of data, while CPUs can run the operating system and perform traditional serial tasks [7].

This is the reason the heterogeneous computing that brings the best of both CPUs and GPUs is essential to drive faster and more powerful processor designs for new and better experiences. GPU is added to a system as a coprocessor to CPU, which supplements its functionalities for the tasks where CPU overloads. This brings a new world of computing where the compute and bandwidth intensive tasks offload the CPU, and the CPU continues it's sequential execution smoothly.This hardware architecture with different kinds of processors is termed as Heterogeneous architecture.

**1.3.1 Heterogeneous architecture**

A Heterogeneous architecture contains different kinds of processors such CPU and GPU.GPU currently is not a standalone platform, but it is attached to CPU as a coprocessor through a PCI-Express bus so that it will work in conjunction with CPU.The CPU monitors all the GPU activities, the CPU is called a host and GPU is a device.

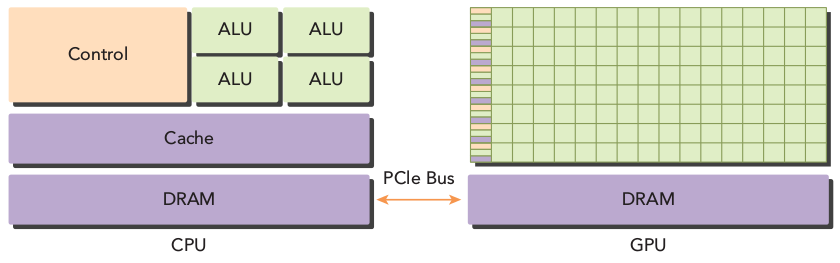


Figure 1.1:Heterogeneous architecture

*(source:Professional CUDA C Programming by John Cheng, Max Grossman, and Ty McKercher )*

In a heterogeneous application, there are two types of codes , a host code that runs on CPU and a device code that runs on GPU,allocating suitable portions for CPU and GPU makes the application perform faster.CPU initializes the heterogeneous application and will manage environment, data and code to the device.

CPU-GPU heterogeneous parallel computing architectures are introduced because the CPU and GPU have complementary attributes and that enable applications to run efficiently using both types of processors.So the sequential parts of the application execute on the CPU and the intensive data parallel parts run on the GPU, as shown in the Figure 1.2. The highlighted portions in the application code of the Figure 1.2 are massively parallel , so this portion is qualified to implement on GPU.While the non highlighted portions are non complex and easily run on CPU.

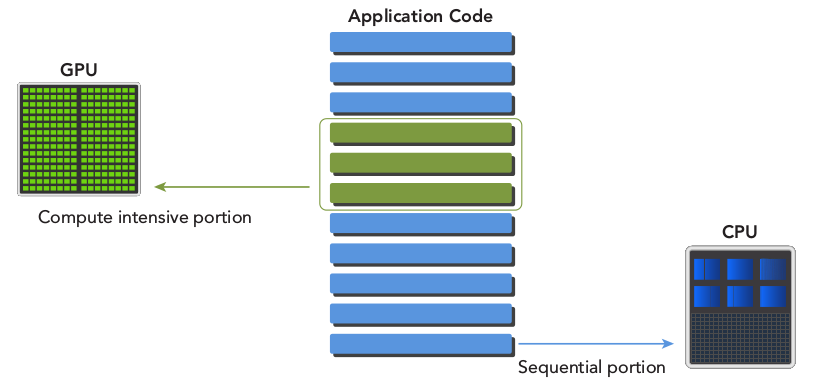


Figure 1.2 Heterogeneous code

*(source: Professional CUDA C Programming by John Cheng, Max Grossman, and Ty McKercher )*

In the heterogeneous architecture the GPU may be integrated into the processor or added externally to the CPU.The discrete GPUs are more common than integrated in the case of desktop PCs.where as in the mobile computing platform integrated GPUs are more common.For this work a desktop PC with NVIDIA GPU is used.NVIDIA GPU is preferred for this work because of its ease to programming.NVIDIA has introduced a language named CUDA C/C++ to program its GPUs, CUDA is expanded as Compute Unified device architecture. More about CUDA will be explained at Chapter 2. The following product families of NVIDIA exhibit the GPU computing platform.

* GeForce
  + It is extensively designed for consumer graphics.
* Quadro
  + This product family is suitable for professional visualization.
* Tesla
  + This family of products mainly used for data center parallel computing.
* Tegra
  + It is designed specially for mobile and embedded computing platforms.

NVIDIA GPUs are classified within the product family according to their architectures such as Fermi,Kepler,Maxwell and Pascal.Each architecture has their own benefits from the previous architectures.In this work Kepler architecture NVIDIA GPU is used.The selection of suitable GPU platform is based upon the following important features,

* Number of GPU cores
* Memory size

A term is coined for using GPU for general purpose programming,which is GPGPU(General purpose Graphics Processing Unit).GPGPU is supported through frameworks such as OpenCL, CUDA etc. CUDA is predominant now a days because of its simple and efficient APIs (Application Programming Interfaces).

**1.3.2 NVIDIA GPU architecture**

As shown in the Figure 1.3, the NVIDIA GPU contains an array of streaming multiprocessors (SM), each SM is capable of running thousands of concurrent threads. The SM in the GPU is a set of processors. A thread is a sequence of instructions to be executed on a processor. A group of threads are called a warp, each warp executes per each instruction cycle in an SM. Instruction level parallelism is extracted by pipe-lining the instructions in each thread. There is no branch prediction and speculative execution in GPU.

NVIDIA GPUs use different types of memory for performance improvements in the Data movement.The shared memory used is local to the block of threads and has very less memory transaction cost.The texture and constant caches are useful for special type of applications and will move the data very fast.

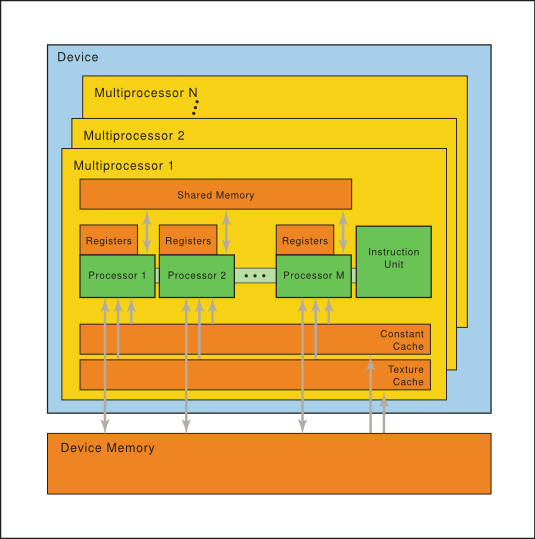
****

Figure 1.3 Hardware model

By Using CUDA platform one can access all the NVIDIA GPU features explained above. CUDA is becoming more significant now a days because of its simple interfaces,and is evolved as a Heterogeneous computing platform that enables the users to easily write heterogeneous programs on to NVIDIA GPU’s.

**Chapter 2 : CUDA programming model**

CUDA is a general-purpose heterogeneous parallel computing platform and programming model that uses the parallel compute engine in NVIDIA GPUs to perform many complex computations in a more efficient way.NVIDIA GPUs are accessible through CUDA platform, CUDA is accessible through CUDA-accelerated libraries(NVIDIA Performance Primitives), compiler directives, APIs (application programming interfaces), and extensions to standard programming languages such as C, C++, Fortran, and Python.This work focuses only on CUDA C.

CUDA C is inherited most of the ANSI C attributes and contains the handful of features to handle parallel programming on NVIDIA GPUs. Some of its features include managing memory, device and tasks for parallel computing.The unique feature of CUDA is its scalable programming model that enables programs to transparently scale their parallelism to GPUs with varying numbers of cores.There are two driver APIs in CUDA to manage the GPU device and organising threads .

* CUDA driver API
* CUDA Runtime API

The device code and host code are separately compiled with the suitable compilers.he host code is compiled by ANSI C compilers, and NVIDIA CUDA nvcc compiles the device code.The device code contains *kernels* which are CUDA C extensions for data parallel functions.During the link stage CUDA runtime libraries added for kernel procedure calls and explicit GPU usage. Programming models act as a bridge between an application and its implementation on computer architecture. The CUDA programming model provides special features like organizing threads on the GPU and a way to access GPU memory.

Parallel computation can be viewed in different levels for a programmer, they are

1. Domain level

At this level, the main concern is to decompose data and functions in an algorithm such that the problem is solved efficiently and correctly in a parallel environment.This level is crucial for every parallel implementation.The motion de-blur algorithm illustrated in this work contains a handful of parallel operations, all these are exposed when the Domain level inspection is performed.The motion deblur algorithm includes gradient operation on the input image, FFT calculation, IFFT calculation, deconvolution operations which are targeted portion for parallel computing in the algorithm.

1. Logic level

When programming for a problem in a parallel environment, our focus is to organise concurrent threads and calculation to solve the problem.In C parallel programming this can be done by using pthreads or openMP techniques.In motion deblur algorithm, the gradient operation is performed in parallel by taking one thread for one output and each thread works on two elements.

1. Hardware level

The efficient mapping of threads to the available cores gives the best performance.In CUDA the thread mapping is done implicitly by considering the programmer's view of parallelism.

**2.1 CUDA development cycle**

Developing an efficient software application is not easy even though we have plenty of resources available for execution.For the purpose of improving application performance, NVIDIA has suggested a development cycle.The development cycle is a repetitive process where refinement to the previous implementation happens and it makes the application perform better.It contains four stages, they are Asses, Parallelize, Optimise and Deploy [10].

**2.1.1 Assess**

In this stage algorithm or code is analysed such that we come to know the most time-consuming parts.Targeting specific portions of the algorithm makes easy work around to start developing the application.In case working with the already developed unoptimized code then time profiling the code gives the suitable parts to parallelize.

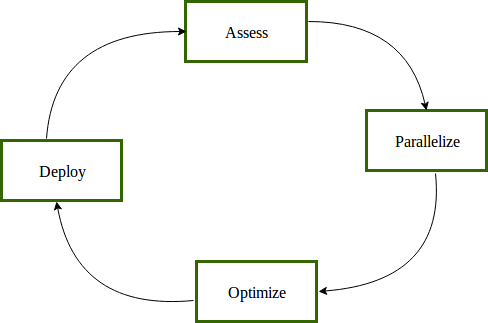


Figure 2.1 CUDA development cycle

**2.1.2 Parallelize**

After Identifying the portions for GPU implementation, the next task is to identify the parallelism in those portions and starting a design task for implementation.It is a very crucial stage and challenges the developer.

**2.1.3 Optimize**

The performance of the application can further be improved by optimising the parallel implementation.This can be a repetitive process of finding an opportunity, applying the optimisation, testing the optimisation and checking the speed achieved and repeating the same process again.Optimisation can be overlapped computation and data transfers, coalesced memory access, reducing global memory access..etc.

**2.1.4 Deploy**

Before finding more hot spots in the algorithm or code the partially palatalized code will be considered for production.This makes the developer risk free and profits the user for his investment by providing an evolution version.

**2.2 CUDA Memory Hierarchy**

One of the unique characteristics of CUDA is its exposed memory architecture that which enables users to use the suitable memory for their application.In CPU memory hierarchy the L1 and L2 caches whose memory transaction cost very less are non-programmable while in CUDA different fast memories are user programmable. Figure 2.2 illustrates different memories on an NVIDIA GPU.

CUDA_MEM_hierarchy

Figure 2.2 NVIDIA Memory architecture

The programmable memories in CUDA are,

1. Registers
   1. Registers are fastest memory in the GPU ,a variable declared in a kernel is expected to store in a register,and these are private to each thread.
   2. The life time is the kernel execution time.The number of register variables used in a kernel effects the occupancy of the kernel.The kernel occupancy is the resource utilization of that kernel.
   3. If a kernel uses more number of registers than the hardware limit, the additional registers will spill over to local memory.The compiler option ”-maxregcount” will limit the maximum number of registers used by all kernel.
2. Shared memory
   1. Shared memory is on-chip so it the memory access cost very low when compared with local and global memory.The CPU L1 cache and shared memory are comparable but shared memory is programmable.
   2. The shared memory is local to each thread block and enables inter thread communication.Shared memory is limited to each SM,so over usage of it limits number of active warps.
3. Local memory
   1. The register variables which can not have a register space to be accommodated are eligible to occupy in a local memory.
   2. Local arrays referenced with indices whose values cannot be determined at compile-time will be allocated in local memory.
   3. Variables mapped to local memory will reside in the global memory ,so the memory access is more costly (high latency and low bandwidth).
4. Constant memory
   1. Constant memory resides in a device memory and is cached to constant cache in each SM.Constant memory is accessible to all kernels,only a 64KB of constant memory is allowed to declare.
   2. Constant memory is a read only memory.
5. Texture memory
   1. Texture memory resides in device memory and will be cached to read only cache in each SM.
   2. It is optimized for 2D spatial locality, so the threads in a warp that use texture memory can achieve best performance.It is used for window operations on an image such as convolution and filtering.
6. Global memory
   1. This is the highest latency memory and it is available throughout the entire application.This is the most commonly used memory.
   2. Global memory resides in the device memory and it is accessible via 32-bit,64-bit,128-bit transactions.
   3. To get the optimal performance of the application the memory transactions should be aligned properly.

**2.2 Memory Management in CUDA**

CUDA kernels operate only on the device memory.CUDA runtime provides several functions to manage memory effectively as shown in Table 2.1. Each of the below functions listed in Table 2.1 returns an error code after invocation which is useful to handle errors effectively.

|  |  |
| --- | --- |
| CUDA C functions | Description |
| *cudaMalloc* | Allocates memory |
| *cudaMemcpy* | Transfers memory across host and device |
| *cudaMemset* | Sets the memory with a given value |
| *cudaFree* | Deallocates the device memory |

Table 2.1: CUDA memory management functions

All the functions are called from host, but they manage memory at device. So CPU has full control over device, as can be seen from Figure 2.3.

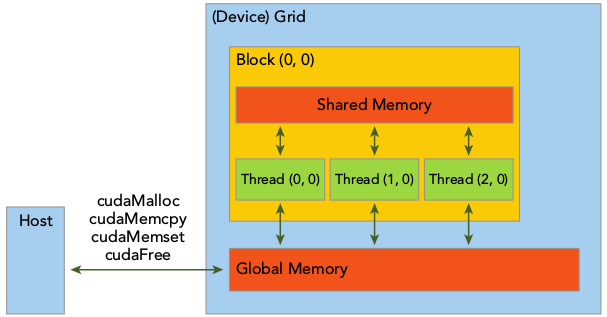


Figure 2.3 Memory management in CUDA

**2.3 Organizing Threads in CUDA:**

When host invokes a kernel function, A large number of threads are created on device which work on the statements specified in the kernel.CUDA follows a thread hierarchy abstraction which provides users to organize threads efficiently.The is two stage thread hierarchy which is decomposed into block of threads and grids of blocks ,as shown in the below Figure 2.4.

All the Threads created in a single kernel launch are called a grid.All the threads in a grid access the same global memory.A grid is made up of many thread blocks ,A thread block is a group of threads which cooperates with each other using shared memory or block level synchronization.

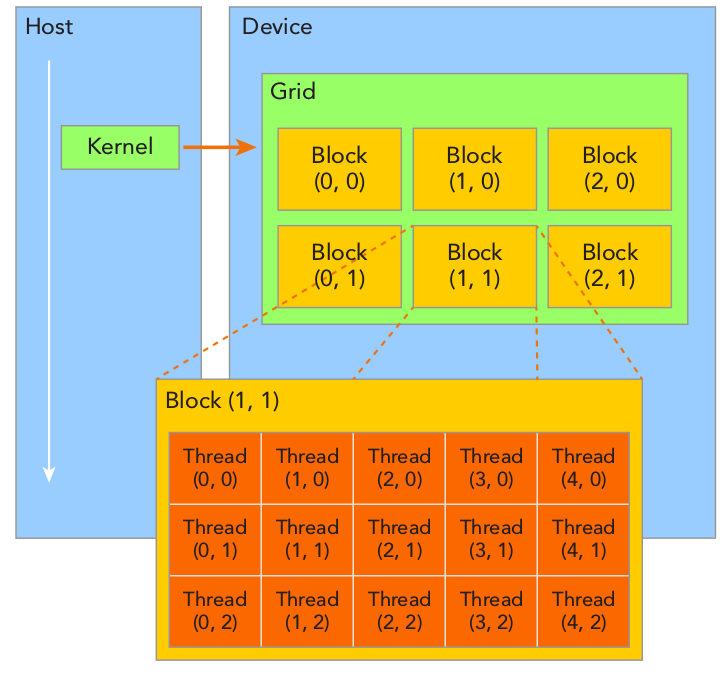


Figure 2.4 Thread-block hierarchy

Each thread is distinguished using two unique coordinates, they are *blockIdx* and *threadIdx* which are assigned to each thread by CUDA runtime . The *blockIdx* and *threadIdx* are of type *uint3* which is derived from basic integer type and comprises three unsigned integers.These three components are accessible with x,y,z extensions as

* *blockIdx.x*
* *blockIdx.y*
* *blockIdx.z*
* *threadIdx.x*
* *threadIdx.y*
* *threadIdx.z*

The grid and block dimensions are specified by the inbuilt variables *blockDim* and *gridDim* which are of type *dim3*.When defi ning a variable of type dim3 , any component left unspecifi ed is initialized to 1.Each component in a variable of type dim3 is accessible through its x , y , and z fi elds, respectively, as shown below

* *blockDim.x*
* *blockDim.y*
* *blockDim.z*

Setting of block and grid sizes is crucial,first we will decide the block size, then we calculate the grid dimension based on the input data size and block size.The block size depends on the performance characteristics of the kernel and on the available GPU resources.These are practiced when implementing the motion deblur algorithm in this work.

**2.5 CUDA programming structure**

The key element in the CUDA programming is the kernel which is the part of the code which runs on the GPU.Implicitly the CUDA manages scheduling the kernels on GPU threads.The main aspect in CUDA programming is memory access,the memory on the CPU is explicitly copied to the GPU memory.The CPU memory is termed as *host memory* where as GPU memory is called as *device memory*.

The host operates independently to the device,the control is returned after a kernel is launched, CPU can perform the sequential tasks.The GPU and CPU activities can overlapped since most of the GPU calls are asynchronous.The host code is in ANSI C and device code is in CUDA C,the different types can be integrated into a single file or can be put into a different files.A typical CUDA program flow is as follows,

1. Allocate Memory on the device
2. Copy data from CPU to GPU.
3. Invoke kernels to operate on the data available in the GPU memory.
4. Copy data back from GPU memory to CPU memory.
5. Deallocate device memory

The program in Appendix A illustrates a matrix addition, two matrices of same size are taken and are added. All the elements in one matrix are added to respective elements in another matrix in parallel with the help of parallel threads using CUDA.The number of parallel threads used in this sample program are equal to the size of matrix,so each thread operates on each addition operation. The matrix Addition program performs matrix addition on GPU and verifies the same on CPU.The steps to write a program for CPU-GPU environment are followed here.When we profile the above program the GPU takes around 0.25 *milli seconds* (excluding memory copies) where as CPU takes 1.39 *milli seconds*.For now we are not considering the latency for memory copies to and fro from GPU, anyway these latencies will be hidden when we are working with large data.

The computer vision and image processing algorithms works on large data as the input being is an image(colour or gray),so GPU enabled environment is most suitable to implement these algorithms.In this work Adaptive Image contrast enhancement,motion deblur algorithm are implemented using CUDA C language on NVIDIA GeForce GT640 GPU. The Autonomous drive algorithms like temporal denoising and fog rectification with denoising are implemented using multi-core CPU(ARM A57).And stereo disparity algorithm is also implemented to extract the hidden capabilities of GPU like CUDA intrinsics,streams, etc.

**Chapter3: Porting of Image processing and Computer vision algorithms**

The image processing and computer vision algorithms are useful in defense, entertainment, digital security,etc. The data being processed in this area is huge. High performance implementations of these algorithms are needed to run them quickly and accurately in real time. One of applications this work is aimed at is processing data coming from UAV(Unmanned Aerial Vehicle). The data will be sent to Automatic Ground Control System(AGCS) from UAV. The hardware and processing will be done at AGCS, and appropriate action will be taken for any intrusions in the surveillance area.

**Components in the AGCS**

The components in the and the target hardware are shown below. The Figure \* shows the list of different hardware connected together to form a High performance computing platform. The Single Board Computer (SBC) acts as a master and other hardware works in conjunction with SBC. From the Figure we can see a GPGPU hardware block, this work is aimed at this hardware block. AGCS follows a client-server

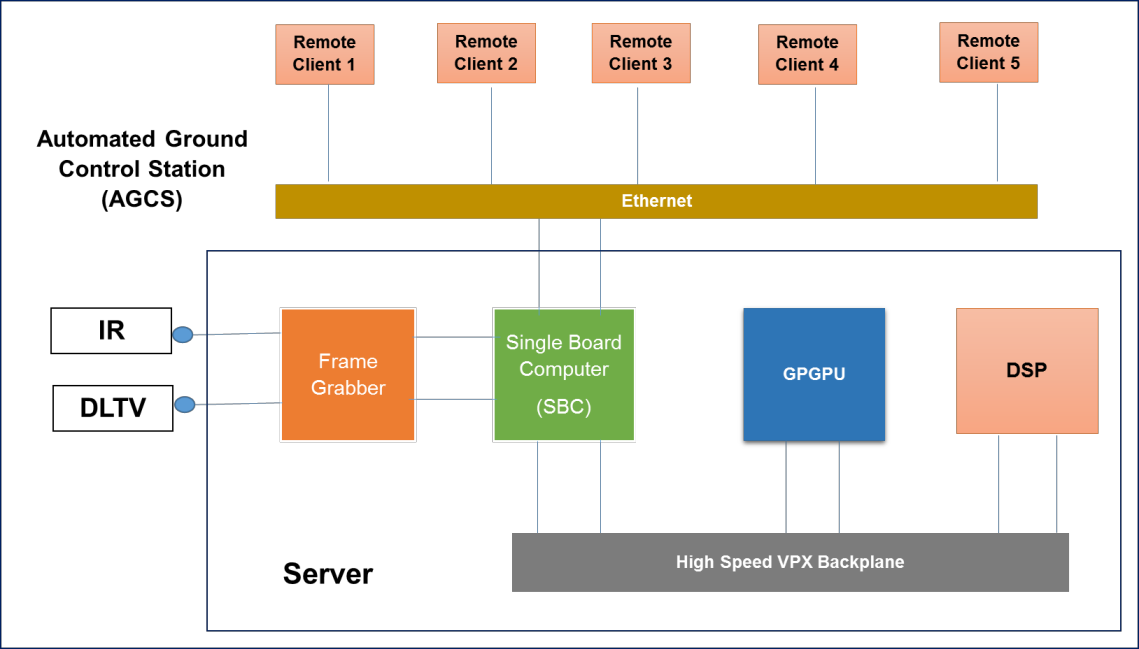


Figure : The different hardware components in automated ground control system

**3.1 Adaptive contrast Enhancement**

Contrast enhancement is used to increase the contrast of an image for better visual appearance. Histogram equalization can perform this task well, but it not immune to noise, it also enhances the the noise in the image. So adaptive contrast enhancement algorithm is developed to avoid the noise component amplification.

**3.1.1 Algorithm Design**

* The first stage in this algorithm is, converting the input image with RGB model to HSV colour model.

/\*Equation\*/

* Then for the luminance component V histogram(Hv(xk)) is calculated.

/\*Equation \*/

* The luminance histogram is smoothed with a gaussian kernel.

/\* Equation \*/

* Selection of Luminance Distributions, the average differences are employed to find the major peaks and valleys. From the output of this difference the positive to negative crossover gives a peak while the negative to positive crossover gives a valley.

/\*Equation average differnce in x direction\*/

* Piece-wise linear transformation for k-1 line segments is, Tk-1(x)=?.

Yk(x) = ?

From the luminance distribution we can get the information regarding peaks. Let the number of luminance distributions are k, i.e {p1,p2,..,pk}(peaks are represented by p), then number of line segments are k+1. In between two peaks there exist a valley, so the distribution has {v0,v1,....,vk} valleys. These valley information is used as the input {x1,x2,...,xk} to the PLT.

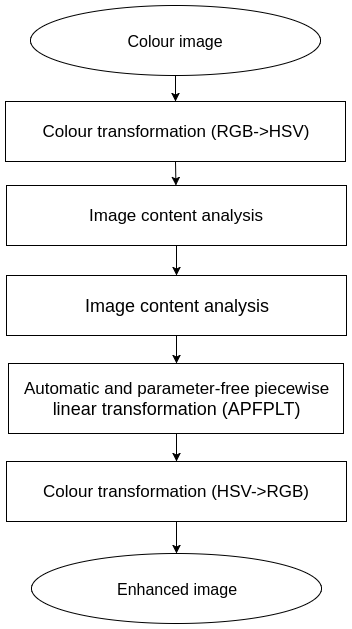


Figure 3.1 Adaptive Image contrast enhancement algorithm flow

**3.1.2 GPGPU Implementation**

3.1.Histogram calculation

* + Histogram calculation is finding the frequency of pixel intensities.This can be done sequentially using *atomicAdd* operations on GPU since a good amount of parallelism can’t be achieved here.Threads equal to image size are created with 1024 per each block and the total number of blocks are decided by the image size like the one shown in the Figure 3.7. When implementing this block-thread configuration yielded good speed up.
* Histogram smoothing
  + Histogram smoothing is done by performing a Gaussian filtering.the next step is the finding of luminance distribution in the histogram, luminance distribution finds the number of valleys and peaks in the histogram.In this step found no scope for parallelism, so it is implemented to run using single thread.
* Piece-wise linear transformation (PLT)
  + After having the required information from histogram soothing, the PLT is implemented on GPU with full parallelism.Since, there is no dependency on other pixels all the created threads works concurrently.A thread-block configuration explained for histogram calculation is used here also.

**3.1.3 Results and Discussions**

The above algorithm is implemented on a system with GeForce GTX 640. And it is tested against different image data and good speed ups are observed. Figure 3.2 shows the input and output of the adaptive contrast enhancement algorithm.



Figure 3.2 input(left) to the Adaptive contrast enhancement algorithm and its respective output(right).

|  |  |  |  |
| --- | --- | --- | --- |
| Image Resolution | CPU(in ms) | GPU(in ms) | Speed Up |
| 640 X 480 | 18.735 | 3.237 | 6x |
| 1280 x 720 | 55.27 | 6.173 | 9x |
| 1920 x 1080 | 104.54 | 11.745 | 9x |

Table 3.1 Timing performance statistics of Adaptive contrast enhancement algorithm.

The Table 3.1 gives the timing performance statistics of the algorithm, it contains the profiling data on CPU also to calculate the speed up. The speed is the ratio of algorithm processing on GPU to the CPU alone time. From the time profiling results we can observe that as long as there is a huge data to process, the speed is also high. Data parallelism benefits us when there is a vast amount of input data.

**3.2 Motion De-blur algorithm**

Restoration of blurred images is useful in consumer level photography,medical imaging and astronomy. One of the reasons for the Blur in an image is due to the motion of the camera , and is called the motion blur . Motion blur in images happens when there exist a relative motion between the imagery(camera) and an object. Here in this work, only blurred images with camera motion are considered and the object(to be captured) is assumed to be static.

**3.2.1 Motion blur**

Blur distorts the details in the image, the sharp image can be restored from the blurred image if we know the blur kernel.Removing blur from the image involves in finding the blur kernel (Point Spread Function) and deconvolving it with the burred image.The process of removing blur from the image is called deblurring.The deconvolution is an important stage, and the deblurring methods are categorised based on the deconvolution methods.The deconvolution methods available are non-iterative weiner algorithm[\*] and iterative Lucy-Richardson, etc...

**3.2.2 Algorithm Design**

The blur in the image can be easily removed if we know the blur kernel , so in this algorithm the first stage is to find the blur kernel and later deconvolve the blurred image with that kernel.

The algorithm is organised as follows,

1. Blur kernel estimation
2. Deconvolution

**3.2.2.1 Blur kernel estimation**

In finding blur kernel there are two stages, calculating blur angle and calculating blur length.

In calculating the blur angle following are the different stages.

* Gradient of the Image

Gradient is the directional change of a scalar; here the gradient is used to find the directional change in the image intensities. The gradient of the Image is simply the edge detection, we will use this data to do the remaining process, generally the edges contains useful data about blur parameters, so processing only on the edges is more easy than the entire image.

* Power spectrum calculation

Power spectrum for the gradient of the image contains a regular structure which has a relationship with the length and angle of the blur in the image. With the properly filtered power spectrum ,the length and blur can be found accurately. The power spectrum is shifted such that all the zero frequencies are aligned at the center of the image.

* Bandpass Filtering

The band pass Butterworth filter can be designed by cascading two Butterworth filters, a Low pass and an High pass. First a Low pass filtering is done on the input and then a High pass filtering follows.The discrete form of Butterworth low pass filter is found by using the below formula



* Radon Transform

The Radon transform operates on the filtered power spectrum and finds the parallel stripes from the pattern in the power spectrum. The Radon transform is used to find the projections of the data at a given angle and a vector will formed by adding the column values,this is continued for several angles.The angle used to form a vector which contains maximum among all the vectors is the blur angle.In such a vector except for the direction of stripes crests and valleys of the projections cancel out so a maximum value in that projection vector refers to the blur direction.

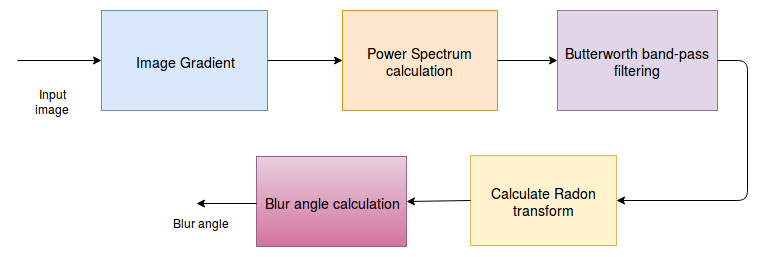


Figure 3.3 Calculating blur angle

The calculation of blur length depends on the blur angle computation.

* Estimate the Target function
  + The spatial domain image data is converted into spectral domain and a log transform is applied which is then converted back to spatial domain.The converted data is rotated with the estimated blur angle and a vector is formed by averaging all the columns in the rotated data.This vector is called target function which contains the information regarding blur length.
* Blur length calculation
  + Once the target function is formed then finding the blur length is easy.The distance between the maximum and the nearest minimum to that maximum in the target function has a inverse relationship with the blur length.The target function is as shown in the figure 3.x also illustrates the blur length calculation.For an image of MxM size the blur length is calculated as follows.
  + Blur length=M/d;

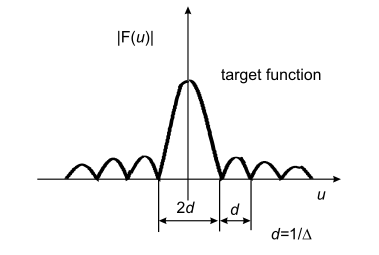


Figure 3.4 Target function

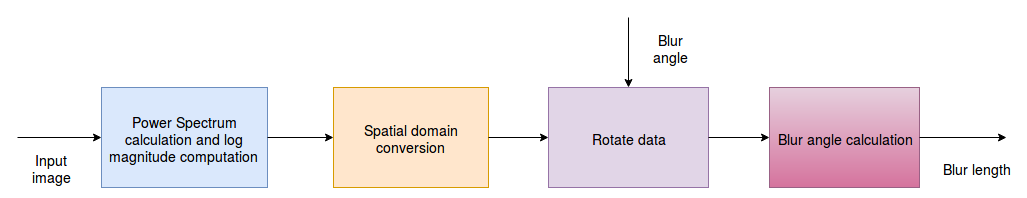


Figure 3.5 Calculating blur length

* Blur kernel estimation
  + With the known blur length and blur blur angle the point spread function(PSF) can easily found.Since the generation of PSF takes a very time on CPU so it is not a targeted portion for parallel implementation.It is kept to run on CPU.

**3.2.2.2 Deconvolution**

The deconvoluton of blurred image with the estimated blur kernel gives the de-blurred image.The sparse method based deconvolution is used in this algorithm.It contains a series of convolutions with the static kernels and followed by a blur kernel.The convolutions will iterate multiple times for estimating the correct input.

* The deconvolution contains a series of convolutions with the following static kernels and the estimated blur kernel.

Kernel1 = [-1,1]

Kernel2 = [-1,-1]

Kernel3 = [-1,2,-1]

Kernel4 = [-1,1,1,-1]

* Three types of convolutions are used in this stage , they are *same,valid* and *full.* Same convolution yields the output with size equal to the input dimensions
* The valid type of convolution yields only the valid pixels, i.e at pixels where there are insufficient pixels surround to form a window are avoided to form a output.
* For example for an image of size 682x482 working with a kernel of 3x3, the output is 640x480. The output size here can be computed by using the kernel size.
* In valid convolution for a 3x3 kernel, the output size in every dimension is reduced by 2 (kernel size-1).
* The *full* convolution gives the padded output. For an image of 640x480 with a convolution kernel axb, the output size in x direction is 640+a-1 and in y direction is 480+b-1.

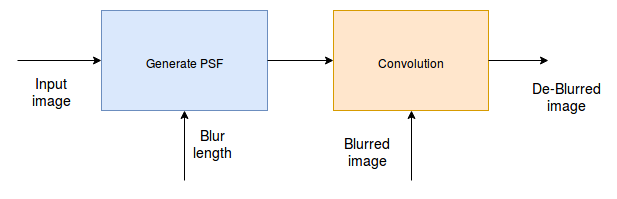


Figure 3.6 Deconvolution

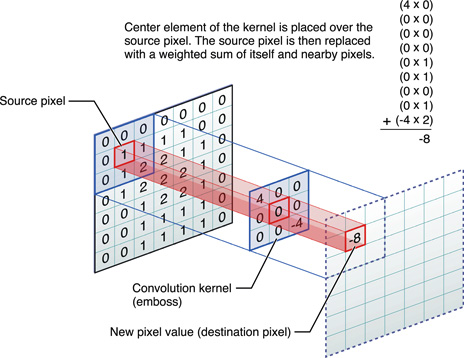


Figure 3.8 Convolution operation

*(source:https://developer.apple.com/library/content/documentation/Performance/Conceptual/vImage/Art/kernel\_convolution.jpg)*

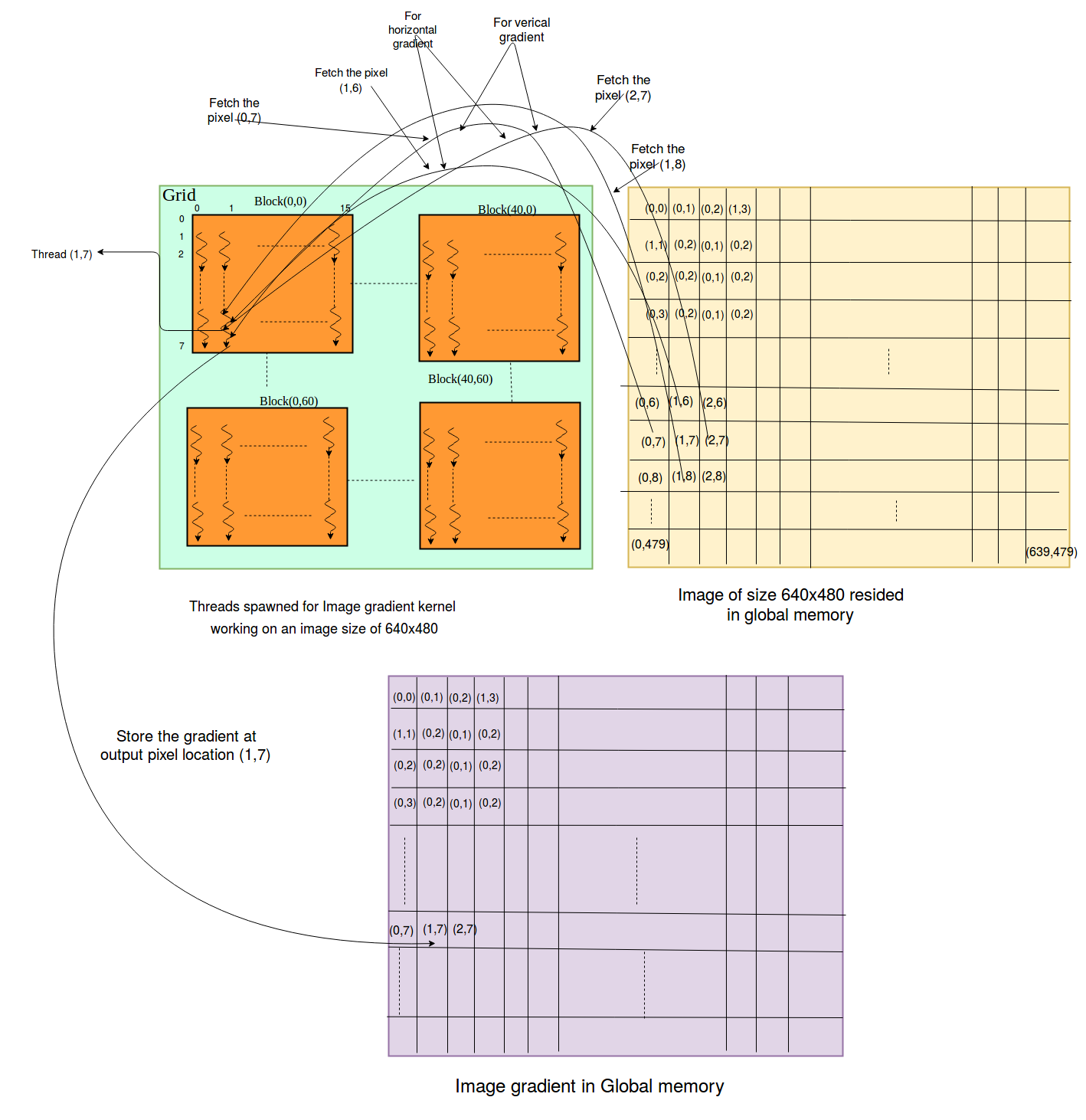
**3.2.3 GPGPU Implementation**

The motion de-blur algorithm is

* Estimating blur kernel

Gradient of the image

* In generating PSF the first step is to find the image gradient. In this step horizontal gradient ,vertical gradient of the image are calculated. The magnitude of horizontal and vertical gradient gives the magnitude of the image.
* The horizontal gradient works on the next pixel and previous pixel in a row. For example if a pixel in the horizontal gradient is at (x,y) then its value depends on the (x-1,y) and (x+1,y) pixels in the input image.
* For a pixel in horizontal gradient at boarder (0,2) then its value is decided by the (0,2) and (1,2) pixels in the input image.
* The vertical gradient works on the next pixel and previous pixel in the column. For example if a pixel in the vertical gradient is at (x,y) then its value depends on the (x,y-1) and (x,y+1) pixels in the input image.
* After the horizontal and vertical gradients are found, the magnitude can be calculated.
* All these three operations produce a single image gradient output. This single output can be computed by a thread at (x,y) in the grid. So output size number of threads can compute concurrently all the image gradient values.
* For example an image of size 640x480 the gradient output is also of same size. As shown in Figure \*, a block of threads with size 16x8 is proposed for computing gradient. In this case a grid with size 40x60 is created to compute all the gradient outputs.



**Power spectrum of the image gradient**

* Power spectrum calculation is implemented using a standard CUDA library *cufft.*
* The power spectrum calculation is explained from the Figure \*.

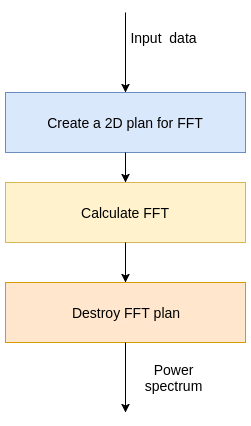


Figure : Using CUDA *cufft* library for calculating FFT.

**Butterworth bandpass filtering**

* Butterworth bandpass filtering is applied on the power spectrum computed from the above step. It removes the unwanted frequencies from the spectrum.
* In this operation first a low pass filter is applied on the spectrum with a cut-off digital frequency of 3 (w). Then high pass filter is applied on the spectrum with cut-off digital frequency of 2.
* Both the filters are joined to form a band pass filter.
* The n in the equation 1.1 is the order of the filter. In this case the filter is of order 5.
* From the equation 1.1, the frequency w(omega) is calculated using the pixel location. For a filtered output at (a,b), the frequency is the square of euclidean distance from the last pixel in the input image.
* Each output produced in the filtering is independent of the other output. An output at (a,b) can be produced with a thread at (a,b) in the grid.
* As shown in the Figure \*, for a spectrum of 640x480 the output is also of same size. A block with 16x8 threads is considered, and a grid of 40x60 is formed to complete all the outputs.

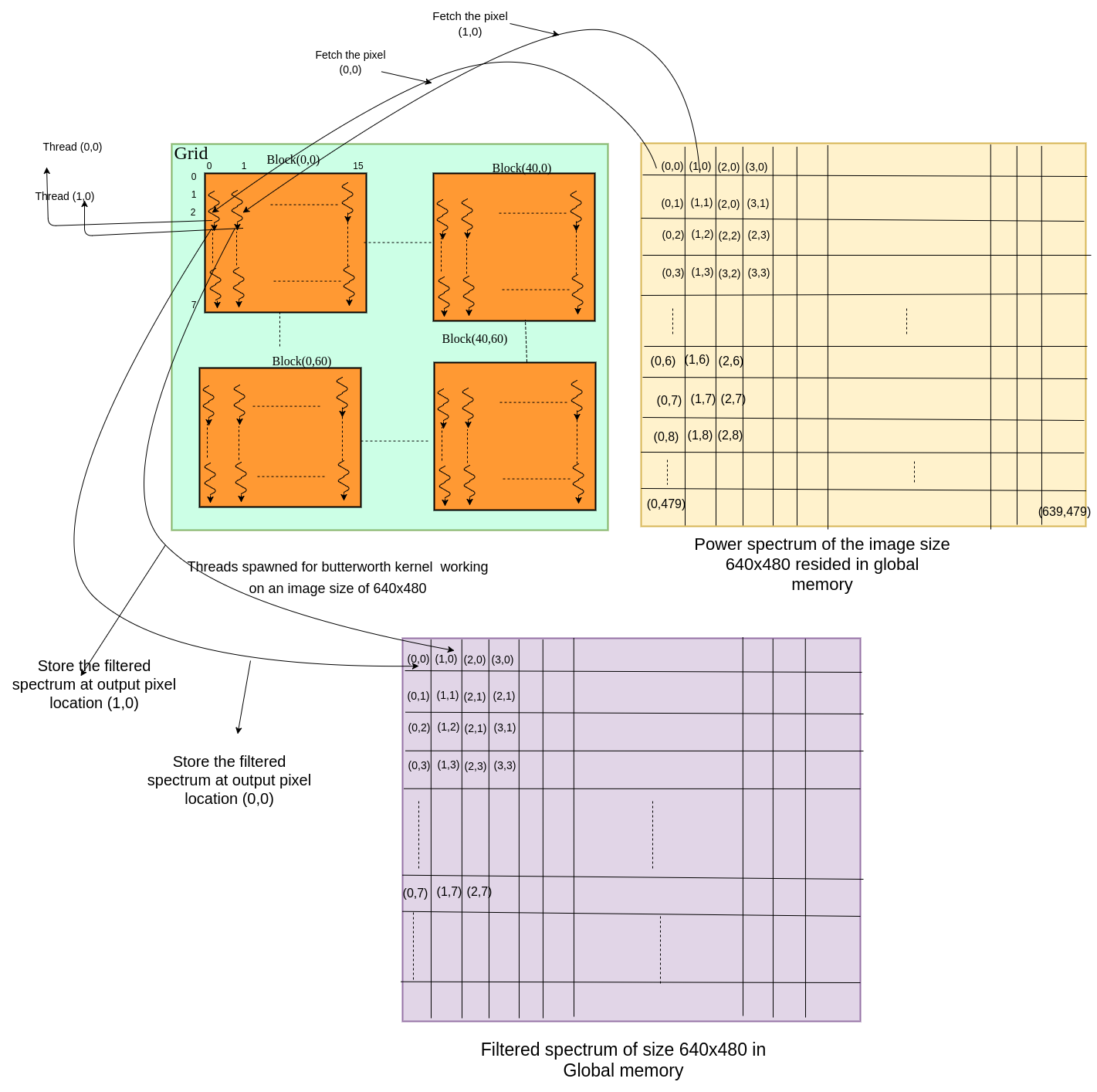
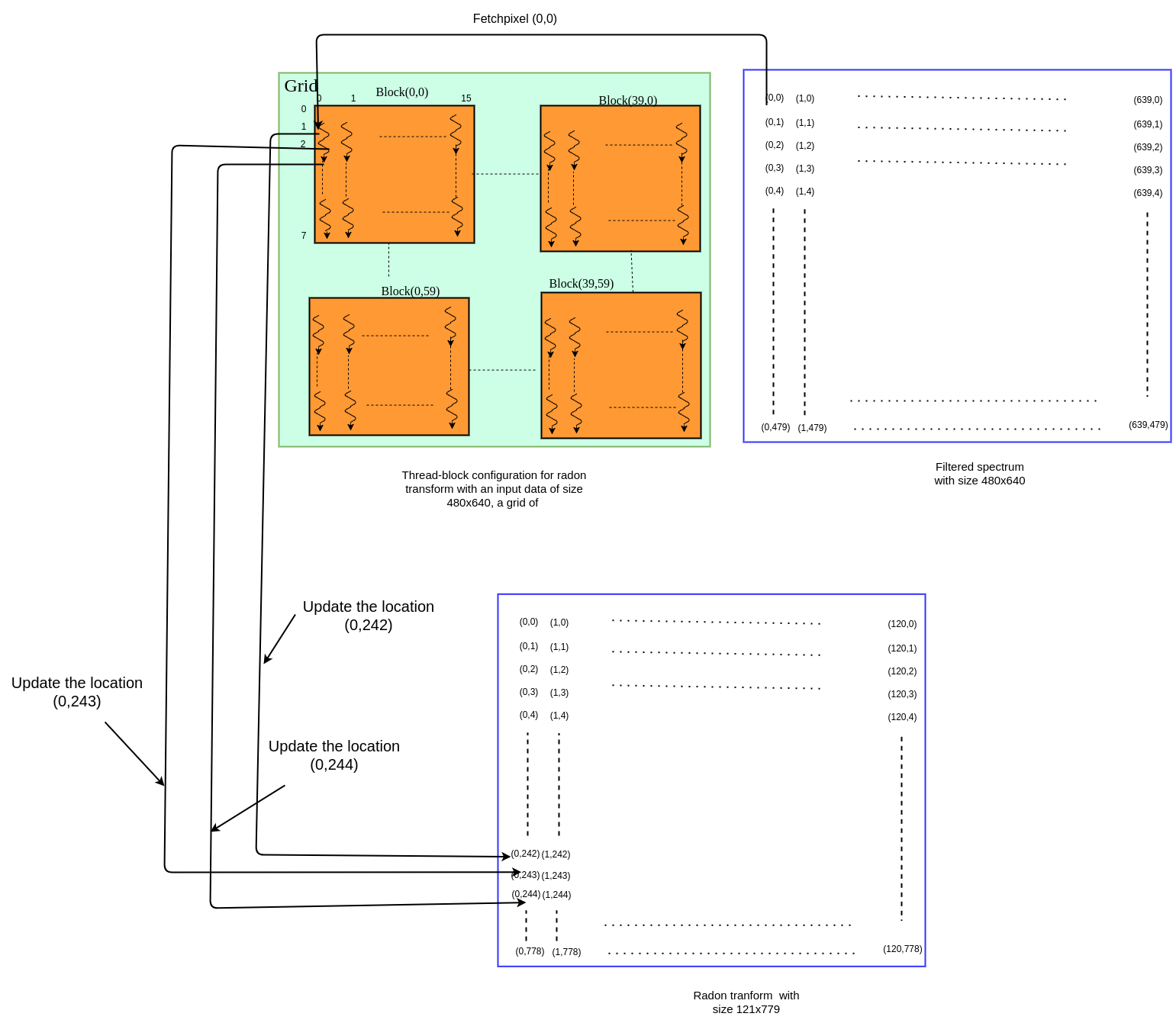


Figure : Butterworth filtering CUDA kernel thread operation

**Radon Transform**

* The Radon transform works on entire image and finds the projections for each angle. Radon transform will be computed for 120 angles in this algorithm starting from angle 30 to angle 150.
* The output size for one angle is found from the dimensions of the data. For an input of size 640x480, the output size for an angle is 1x799. To accommodate for 121 angles output, the output size is 121x799.
* For each pixel 4 projections are found with a 0.5 unit distance between the consecutive projections. Each projection updates output data at two locations. One at the projection and other at the 121(total number of angles) locations away from the initial projection. So a single pixel contributes at 8 locations of the radon output. These locations might intersect.
* Single pixel distribution to 8 locations can be done by a single thread, then here data parallelism can be extracted using data size number of threads.
* For example, for a data of size 640x480, grid of 640x480 threads can concurrently calculate projections and update the pixel data at those locations.
* For this case the pixel data is accumulated at the computed projections, So the accumulation operation is carried out using *atomicAdd* operation this breaks some parallelism by synchronizing threads which are accessing the same location to accumulate the pixel data.
* A block size of 16x8 is considered, and a 2D grid of size 40x60 is created to complete the operation for all pixels.
* As shown in the Figure \*(radon thread access pattern), the input data size is 640x480 then the output data size is 121x799. The thread (0,0) works on a pixel (0,0).
* At an angle of 30o, the first projection location is (0,242), the horizontal 0.5 distant second projection is (0,242). The verical 0.5 distant third projection location is calculated as (0,243) and fourth projection with 0.5 distant in both directions is (0,243).



**Blur length calculation**

* The input image is first converted into frequency domain. This can be done by using standard CUDA fft library cufft , as shown in Figure \*.
* Magnitude of the power spectrum is calculated, and it is log transformed.
* Finding the magnitude and log transform are pixel level operations and does not depend on the other pixels. An output at location (u,v) can be produced by a single thread at (u,v) in the grid.
* As shown in the Figure \*, for a spectrum of 640x480 the output is also of same size. A block with 16x8 threads are found to be efficient, and a grid of 40x60 is created to complete all the outputs.
* The result of log transform is rotated by blur angle. This data rotation is performed using *OpenCV* library on CPU and the blur calculation too implemented on CPU.
* The blur kernel estimation takes very less time on PU and it did not qualify for data parallelism.

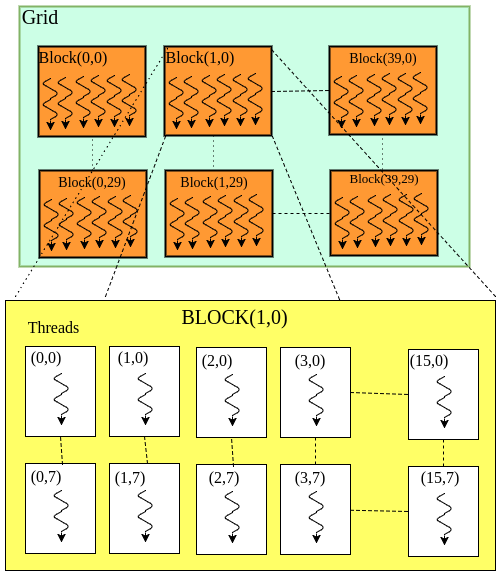


Figure 3.7 Thread-block configuration for radon transform with 40x30 blocks and each block contains 16x8 threads, the kernel works on data of 640x480.

* **Deconvolution**
* As explained in the algorithm, the deconvolution contains a series of convolutions.
* The sequence of operations are explained in the Figure \*.
* Initially the image is padded such that the boarders are extended by half kernel size. Its just a element wise copy operation. Single element copy operation can be handled by a single thread.
* Creating a grid of threads with size equal to padded image then all the elements can be copied simultaneously. 16x16 threads for block considered in this case.
* For an image of 640x480 and a kernel with size 3x3 the output is 642x482. So a grid of 41x31 blocks is created to complete the entire operation.
* The padded image is subjected to valid type of convolution with the flipped blur kernel. A window of pixels from the image multiplies with the respective elements from the blur kernel. The outputs from the multiplications are accumulated to form a single convolution output.
* If a thread can produce an output then the output size number of threads can generate all the outputs simultaneously. But each thread will iterate over multiple times controlled by the size of kernel.
* A block with size 16x16 is considered for convolution operation. For the above padded image with size 642x482 needs 41x31 blocks to complete the entire convolution outputs. This valid operation is shown in Figure \*(convolution kernel).
* For the remaining types of convolutions, creating output size number of threads will complete the convolution operations.
* The subtraction operation From the Figure \* is element wise. For a inputs of size 642x482, this operation can be done on GPU with a kernel with a grid of 41x31 blocks. Each block contains 16x16 threads.
* Calculating factor *rho* and *alpha* needs to add all the elements from the output of subtraction operation. So *atomicAdd* operations are used to add the elements.

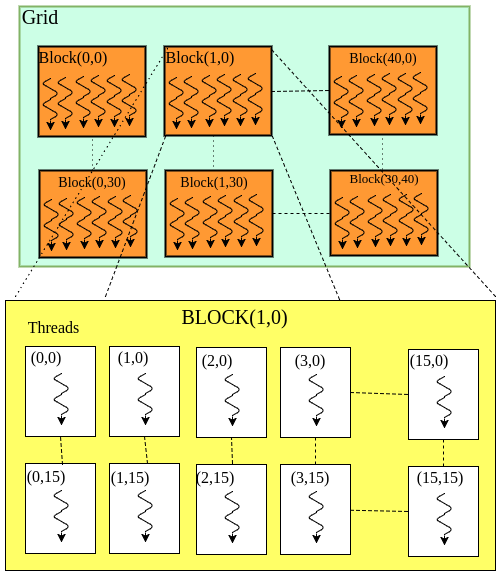


Figure : Deconvolution operation thread-block configuration with 16x16 threads in a block and 41x31 blocks in a grid, working on an image of size 642x482.

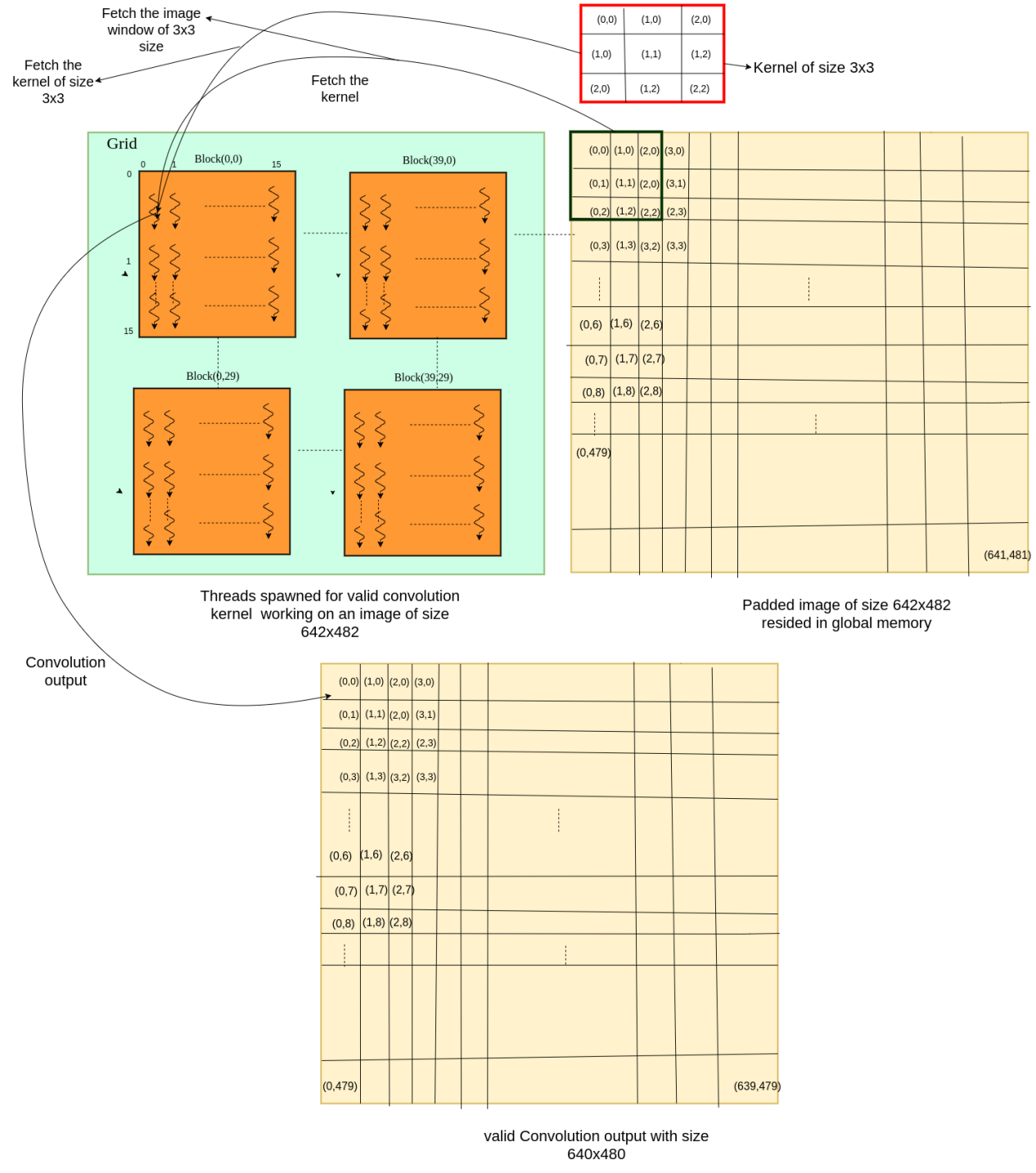


Figure : Valid type of convolution operation on GPU, threads access pattern. For an image of size 642x482 and a convolution kernel of size 3x3.

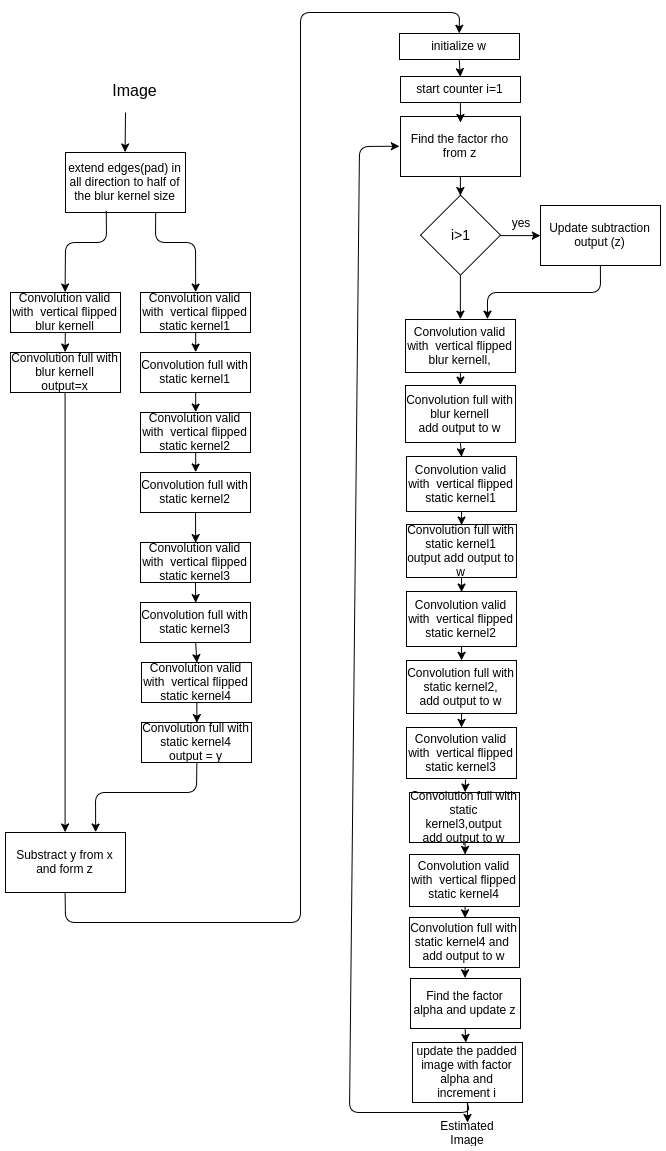


Figure : Sequence of operations in Deconvolution

**3.2.4 Results and Discussions**

The ported algorithm is verified against different inputs with different sizes, Table 3.2 shows the timing Performance of the motion de-blur algorithm for different sizes of input.From the table 3.2 it is obvious to note that whenever input data is more then the speed up is more, so there is a more possibility to process this huge data in parallel.

From the Table 3.2 the module radon transform speed up is low.There is a dependency in the data while finding the projections , *atomicAddd* operation are used for this dependency , they are the main cause for less speed up.All the remaining modules data parallel so, a good speed up is observed in all the cases.All these speed up are observed without any use of texture or shared memory.Infact the there is a possibility to use shared memory in some modules, the Gradient image works on two elements and in this current scenario those two are fetched from global memory, to save one memory transaction we can use shared memory to contain data.This can be applied to convolution kernel also where a window of pixels are used for a single output.



Figure 3.10 Motion de-blur algorithm input(left) output (right)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Module | CPU(in ms) | | GPU(in ms) | | Speed Up | |
| 480p | 1080p | 480p | 1080p | 480p | 1080p |
| Gradient of the image | 14.50 | 90.94 | 0.517 | 2.32 | 28 | 39 |
| Power spectrum calculation | 18.24 | 154.06 | 1.261 | 8.813 | 17 | 19 |
| Butterworth bandpass filtering | 22.82 | 135.29 | 2.39 | 3.722 | 10 | 36 |
| Radon transform | 16.52 | 55.12 | 3.85 | 12.05 | 4.3 | 4.5 |
| Deconvolution | 320.5 | 3246.99 | 29.86 | 194.58 | 11 | 17 |

Table 3.2 motion de-blur algorithm performance analysis

**Chapter 4: Porting of Autonomous drive algorithms**

Autonomous driving is a move to the next level driver less transportation.It makes the transportation simpler and easier as the user don’t need to know driving and he don’t need to apply for a driving license.It sophisticates human living with driver less vehicles and also carries a big set of challenges to the technology.As it is a driver less transportation, mainlining risk free deals is a typical task.All the autonomous systems works based on the central theme of Artificial intelligence.Decision making while driving(in the context of autonomous drive systems) without human intervention needs a lot of processing on the huge input data.The autonomous drive system should make a decision very fast so that it gains good control on the vehicle while driving.To make quicker decisions the system should have good hardware support, and all the decision making algorithms should be implemented efficiently, so that they fully utilize the hardware and run very fast.

**4.1 Embedded automotive platform**

Embedded automotive platforms aim to target the applications in vehicles like In-Vehicle Infotainment (IVI), tracking, object detection, etc.

**4.1.1 Rcar-H3 SiP**

Rcar-H3 SiP(System in Package) is announced by Renesas and is not in full production.It is a successor to its previous drive platform Rcar-H2.The Rcar-H3 is fully designed to work for autonomous drive systems.This can be used in vehicles for In-Vehicle Infotainment(IVI) , navigation, object detection, collision estimation, etc. The full hardware details of Rcar-H3 SiP are exposed here.

**4.1.1.1 Hardware specifications**

The Rcar-H3 SiP contains a CPU core with ARM®Cortex®-A57 Quad, ARM®Cortex®-A53 Quad, ARM®Cortex®-R7 Dual lock-step.The A-57 is a powerful CPU core while A-53 is little low when compared to A-57. This combination of high and low CPU cores makes ARM’s big.Little architecture.In this architecture the statements to execute will be streamlined to different cores for processing according to their complexity.The SiP has a external LPDDR4-SDRAM support that will be used to extend the RAM.It has several Audio,video modules which are specifically designed to perform a defined set of functionalities.

A full overview of Hardware specifications is provided in the below Figure 4.1.

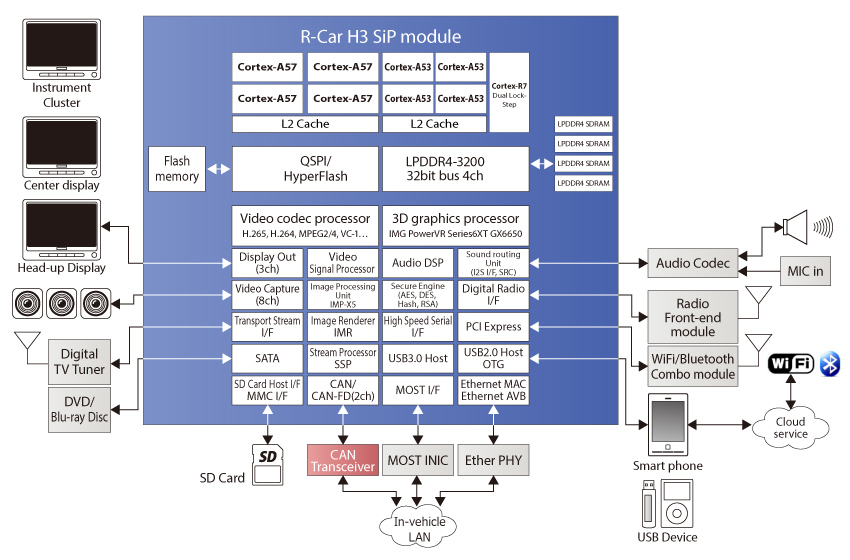


Figure 4.1 Rcar-H3 SiP Hardware block diagram

**4.1.2 Tegra X1 SoC**

Tegra X1 SoC is brought by NVIDIA to compete in the mobile processor line.This SoC contains ARM A-57 quad MPcore CPU and a Maxwell architecture NVIDIA GPU with 256 cores.It contains special video decoders such as H.265, H.264 for rendering HD, UHD, 4K videos.

**4.2 Temporal denoising**

The temporal denoising removes the noise between the video frames . It found uses in conjunction with other image processing algorithms like fog rectification, low light enhancement , etc. This algorithm is used in this work to remove the noise component added during the processing of autonomous drive algorithms.

**4.2.1 Algorithm Design**

To describe the algorithm, An average frame is found using the motion of the frames and current frame, the average frame is used to denoise the current frame using standard procedures[\*]. The algorithm is organised into two parts as calculating the average frame and reconstructing the entire denoised image. The algorithm is shown in the Figure 4.2.

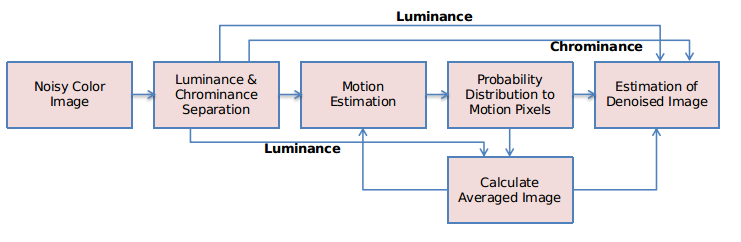


Figure 4.2 Temporal denoising algorithm

**4.2.2 Multi-core CPU Implementation**

In first implementation of the algorithm an entire image is processing on single core of quad ARM A-57 MPcore in Rar-H3 SiP. Since a core contains limited resources, and processing an entire image has given less performance and the time of the algorithm increased considerably. So the idea is to fully utilise the four core performance of the processor. So to achieve this the image is divided and is distributed to process in parallel on the four cores , as shown in Figure 4.2. This ensures the data level parallelism, every core processes quarter of an image thus expecting all the four cores to be busy in processing a single image. At the end the output of all the cores is joined as a single output image.

The ARM A-57 MPcore is programmed using *pthreads* library to accommodate parallel processing on all the four cores. Four threads are created and each thread’s affinity is set to one of the four cores. All the threads are running parallel and the entire processing takes comparatively less time with the single core implementation.

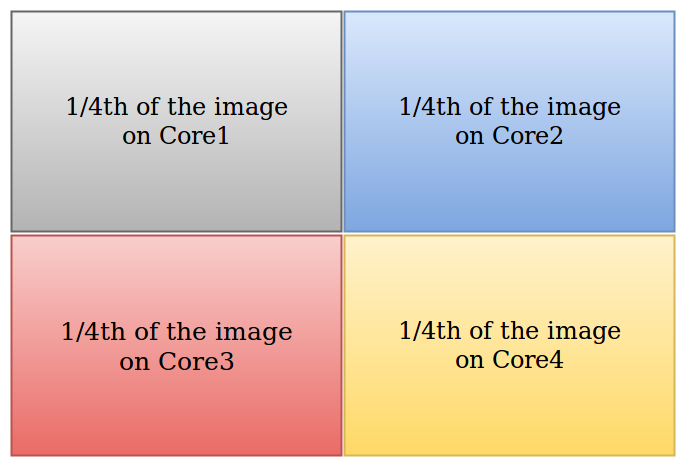


Figure 4.3 Divide the image and process

**4.2.3 Results and Discussions**

The temporal denoising algorithm is the best suited example for multi-core implementation, the performance achieved in this implementation is satisfactory. The initial implementation of the algorithm runs at 4 FPS, while the multi-core implementation runs at 14.71 FPS on the Rcar-H3 SiP ,shown in the Table 4.4 . The Figure 4.3 is shows the input and output to the temporal denoising algorithm. 

Figure 4.4 Temporal denoising input(left) and output(right)

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Resolution** | **FPS** | |
| **Without multi-threading** | **With multi-threading** |
| 1 | 600x404 | 4.5 | 14.71 |

Table 4.1 Temporal denoising performance statistics

**4.3 Fog Rectification with denoising**

Fog rectification is used to remove the fog from fog effected scenes. It is very useful in autonomous driving in winter seasons where the information in the scenes is blocked by fog. The fog rectified image may contain noise, so applying denoising algorithm on the output of the fog rectified image preserves good details of the image.

**4.3.1 Algorithm design**

Initially minimal intensity component in the colour image for every pixel is separated out and is called dark channel.The later operations will be performed on this dark channel. The anisotropic diffusion extracts the fog component from the dark channel.This extracted fog intensities are used to remove the fog from the scene. The Figure 4.5 better explains the block implementation of the algorithm.

Figure 4.5 Block diagram of fog rectification algorithm

**/\*Include denoising block at the end\*/**

**4.3.2 Multi-core CPU Implementation**

The single core implementation of fog rectification algorithm takes much time to run.The Fog rectification algorithm when implemented on hardware is divided into three parts, decoding the image, processing the image (implement algorithm on the image), and displaying the output. In a single core implementation all these will run on single core and due to the limited resources the execution of these portion will take more time than expected.

If we make the three parts to run on different cores of a multi-core processor and maintaining proper communication among them runs the application faster, this is because of the availability of good amount of resources for the each portion to execute.This algorithm is implemented on Rcar-H3 SiP which has an active ARM A-57 MPcore.

Here, All the three portion of the algorithm are dependent on each other, so running them in parallel is not possible. A basic core level pipelining concept is used here. Initially an image is decoded at the processing portion, when the processing stage takes the image for processing then it notifies to decoder to decode a frame, while the processing stage processes the image the decoder on the other hand decodes the image in parallel. After the completion of processing, the display stage takes the image and notifies the processing stage to process next image, so while displaying the processing stage processes the image. This activity goes on and after few cycles all the three stages will run concurrently but on different images, as shown in Figure 4.6.

The available hardware in this work is Rcar-H3 which has an active ARM A-57, the decoding part of the algorithm is made to run on the core 1, the Processing part is provided with 2,3 cores, and finally the display part of the algorithm runs on core 4. Good improvement is observed in the algorithm execution timing performance , the details are enclosed in the results section.

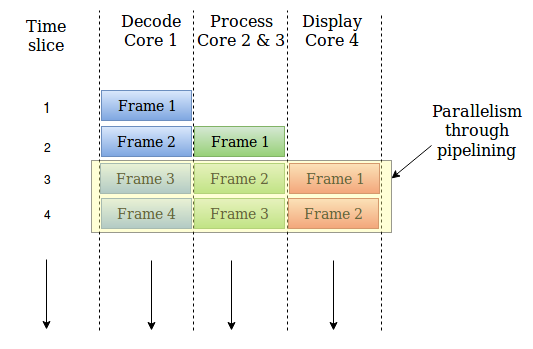


Figure 4.6 Parallelism through pipelining

**4.3.3 Results and discussions**

The multi-core implementation of the fog rectification algorithm runs smoothly on the Rcar-H3 SiP at 23 FPS. The single core implementation of the same is laggy and fall behind expected FPS. The ARM A-57 is a strong MPcore that made it possible to run the algorithm in real time. This algorithm is implemented in conjunction with the denoising algorithm, and achieved good performance there too. The single core fog rectification with denoising runs at 4 FPS where as the multi-core implementation runs at 8 FPS. The results can be seen in the Figure 4.7 and the timing performance is tabulated in Table 4.2.



Figure 4.7 Fog rectification input(left ) output(right)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S.No** | **Resolution** | **FPS** | | | |
| **Without denoising** | | **With denoising** | |
| **Normal implementation** | **Multi-core implementation** | **Normal implementation** | **Multi-core implementation** |
| 1 | 600x404 | 16 | 23 | 4 | 8.5 |

Table 4.2 Fog rectification algorithm result

**4.4 Stereo disparity block matching**

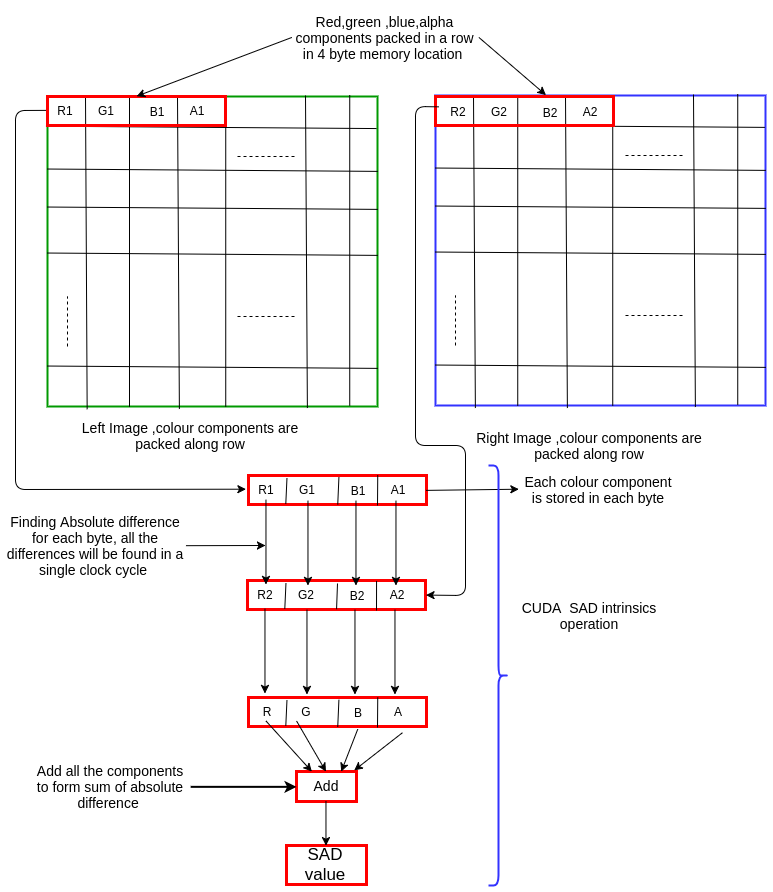
One of the most concerned areas in the autonomous drive is finding the object distance from the vehicle. The solution is to find a disparity map, the disparity map is inversely proportional to the pixel distance from the imagery.The disparity map calculation uses rectified stereo images.

**4.4.1 Algorithm Design**

* In the two rectified stereo images ,one will act as a reference image.In this case right image is reference image.
* Disparity map is finding the level of similarity in the stereo images.For finding the similarity in the images, a block of pixels are considered instead of single pixel.
* The Figure \* shows, a block of 17x17 pixels is used in the left image for finding similarity in the reference image over a search window of 16 pixels.
* The pixel window in the left image is compared against the pixel window in the reference image and a Sum of absolute differences(SAD) matrix is found. And the pixel window in the reference image is shifted by a pixel and a SAD matrix is found again. This continues upto 17 shifts in the reference pixel window.
* The shift location of minimum SAD cost is noted as the disparity value at that pixel.

**4.4.2 Parallel GPU Streams Implementation**

* If we closely look at the SAD operation, for every SAD value the difference values needed to form a SAD value are already available except at that pixel.So, instead of finding difference for each SAD operation, if we find the difference between the two images then it can be used to form SAD cost matrix. This avoids some redundant calculations.
* Finding difference between each pixel of both images is sufficient rather using a window based approach. In the difference matrix summing a window of 17x17 elements centered around a pixel gives the SAD value at that pixel.
* The summation operation to form a SAD matrix is implemented using convolution operation with a 17x17 all ones kernel.Further this kernel is splited into two linear vectors (horizontal and vertical) with a size of 17 to replace the single 2D convolution with two linear 1D convolutions which is called a separable convolution.
* The difference matrix is first operated with the horizontal kernel and later the output is operated with a vertical kernel , this gives the entire 2D convolution output.
* This difference matrix will be found for 16 times, each time finding difference matrix with a pixel shift in the reference image and SAD matrix is also found with separable convolution.
* The shift location of The minimum SAD value among all the SAD matrices gives the best disparity value at that pixel (all the SAD values are mapped to respected pixel locations).
* The difference matrix formation at pixel level is independent of the other pixel, So this is more qualified for data parallelism.The difference operation is performed on all the 3 components (RGB) in a pixel and accumulated to a single value. If each output data element is expected to produce by a thread then a data size number of threads will give the entire difference matrix.A linear grid of 704 blocks with 1024 threads in each block are created to perform the operation,the threads are arranged in 32x32 2D fashion .
* The horizontal separable convolution operation is applied on the difference matrix, since the single convolution output uses multiple input data elements so there is a chance of redundant memory transactions.To avoid this unnecessary memory access,the difference matrix can be stored in a shared memory and will be available to access multiple times with less time cost.A 18x20 (columnsxrows) 2D grid of blocks is used with 32x32 threads in each block.
* So the above methodology implements convolution operation very effectively that it takes very less time compared to the standard implementation.This methodology is used for the other separable convolution too.
* The output of the convolution output is subjected to vertical separable convolution, the output gives the SAD matrix. A 17x20 (columnsxrows) 2D grid of blocks is used with 32x32 threads in each block.
* The SAD matrix is compared against with the previous minimum SAD values stored in a matrix, and best is placed in the minimum SAD value matrix and shift location of best SAD value is also stored in a disparity matrix.This operation runs on a GPU kernel with 667 linear blocks ,each block is having 512 threads.

****

**Figure : CUDA intrinsics operation**

**Texture Memory can be used for some implementations**

* From the above CUDA kernel which performs difference operation, it is evident that there is a possibility to increase speed up using texture memory.
* NVIDIA provides for its GPU a special cache called texture memory for 2D locality referencing. The memory costs are very less when a variable is read from texture cache. The texture memory is read only.
* The images can be cached to texture memory to reduce the memory costs. The texture memory usage is dominant in window based operations on images.

**CUDA intrinsics can be used to perform SAD operations very quickly in a single cycle.**

* In general implementation for a color image the difference is found for each component and all are added. Here exist some parallelism using CUDA intrinsics, since each component takes a single byte to store then CUDA per byte intrinsics will do the difference operation to three components simultaneously and will add all the differences at the end to give a single value.
* The CUDA intrinsics work on 4 bytes memory, So here the three components are packed to a single component with Alpha channel added at end. The alpha channel appended to each pixel will have zero value.
* The CUDA SAD intrinsic operation is explained in the Figure \*.

/\* Include stereo disparity algorithm figure \*/

**Parallel Streams can be used to Implement the independent operations concurrently in the algorithm, Finding the SAD costs for 17 disparity levels is qualified for streams based implementation.**

* A sequence operations that run on GPU in the order which CPU has issued. Operations in the streams can be run in parallel if sufficient device are available for execution.Streams are asynchronous calls in CUDA.
* The close observation on the above implementation reveals that the SAD matrices found in different pixel shifts of the reference image are independent of the other. So, all the SAD matrices can be found simultaneously using streams.
* Streams in CUDA can run different kernels simultaneously, with the limitation that the kernels shouldn’t exhaust the resources of the GPU.
* To implement the stereo disparity algorithm using streams, the number of resources used per each kernel are reduced, making kernels light weight. So that any single kernel will not exhaust the entire GPU resources.
* The kernel launched to perform difference operation takes full utilization of hardware when data parallelism is used such that each output is correspond to each thread.When modified for streams, the per thread load is increased such that each thread is given more work.The initial implementation creates 704 blocks with 1024 threads in each block, it occupies 90% of the GPU resources.On the other hand created 8 blocks with 256 threads in each block ,so that 4 threads works for each row of an image with size 640x533 , the increased work load per thread decreased the occupancy to 13% while increasing the time for execution of the kernel.
* The separable convolution implementation is also modified according to the streams. For these two kernels , image is processed patches wise, each time process a 64x64 patch. The horizontal separable convolution before modification creates a grid of 18x20 blocks with 32x32 threads in each block. And also, vertical separable convolution before modification creates a grid of 17x20 blocks with 32x32 threads in each block.Both the kernels use 90% of the entire hardware individually. So, to run many of those kernels in parallel they are modified such that a grid of 4x4 blocks with 16x16 threads in each block are created, now the modified kernels takes each around 13.5% of the resources.
* With this implementation we can run 8 streams simultaneously on NVIDIA GTX 1070 GPU which has 1920 CUDA cores.

**/\*Use NVVP screeenshots for streams implementation\*/**

**4.4.3 Results and discussions**

The stereo disparity block matching algorithm implementation has given good performance. The algorithm has run on different variants of GPUs to check for its scalability. The below images in Figure 4.5 are input to the stereo disparity and the image in Figure 4.6 is the output of the algorithm. The algorithm runs very fast on GeForce GTX 640 which has 394 CUDA cores.



Figure 4.5 stereo disparity block matching input(left) and input(right) with size 533x640



Figure 4.6 stereo disparity block matching output (size 533x640)

Table 4.3 gives the performance of stereo disparity algorithm on varied image sizes. The image sizes have linear effect on the processing time.

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | Image Resolution (Widthxheight) | Timing performance (in ms) | |
| Implementation without streams | Implementation with streams implementation |
| 1 | 320x240 | 1.75 | 2.3 |
| 2 | 800x600 | 6.14 | 5.6 |
| 3 | 640x533 | 4.7 | 4.68 |
| 4 | 960x720 | 7.5 | 6.8 |

Table 4.3 stereo disparity performance on GeForce GTX 1070 for varied sizes of input image

|  |  |  |  |
| --- | --- | --- | --- |
| S.No | Disparity levels | Timing performance (in ms) | |
| Implementation without streams | Implementation with streams implementation |
| 1 | 8 | 4.02 | 3.31 |
| 2 | 16 | 4.63 | 4.75 |
| 3 | 24 | 5.84 | 5.42 |
| 4 | 32 | 6.81 | 6.49 |
| 5 | 40 | 8.02 | 6.88 |
| 6 | 48 | 8.39 | 7.62 |
| 7 | 64 | 10.00 | 8.90 |
| 8 | 96 | 13.73 | 12.79 |
| 9 | 128 | 17.09 | 16.53 |

Table 4.4 stereo disparity performance on GeForce GTX 1070 for input image of size 533x640 for various disparity levels

From the above Table 4.4 it is evident that a small gain in performance is achieved for the streams based implementation over the general implementation, 8 streams are able to run concurrently.

**Chapter 6: Conclusions and Future Scope**

Many of the high performance parallel implementations are discussed in this work. GPU parallel implementation for image processing and computer vision algorithms Adaptive contrast enhancement, motion de-blur is discussed. A basic approach to implement those algorithms on GPU is presented, and a very expert implementation on GPU are explained for autonomous drive algorithm stereo disparity block matching. In the stereo disparity block matching algorithm discussed about the usage of CUDA intrinsics, parallel streams, shared memory and texture memory.

Not only the GPU implementation, but a parallel high performance implementation through CPU is also mentioned at autonomous drive algorithms temporal denoising and fog rectification. The complete utilization of multi-core CPU is explained, the approaches to extract parallel performance for data dependent operations using pipelining is focused at fog rectification.

The high performance implementations can be extended to the advanced technological areas like deep learning, visual computing etc. Some more areas like finance, marketing, statistical analytics also benefit from the high performance computing.

**References**

1. Tsai, Chun-Ming, and Zong-Mu Yeh. "Contrast enhancement by automatic and parameter-free piecewise linear transformation for color images." IEEE Transactions on Consumer Electronics 54.2 (2008): 213-219.
2. Dobes, Michal, Libor Machala, and Tomas First. "Blurred image restoration: A fast method of finding the motion length and angle." Digital Signal Processing 20.6 (2010): 1677-1686.
3. Tiwari, Shamik, et al. "Review of motion blur estimation techniques." Journal of Image and Graphics 1.4 (2013): 176-184.
4. Professional CUDA C Programming by John Cheng, Max Grossman, and Ty McKercher.
5. “Single image fog removal using anisotropic diffusion” A.K. Tripathi S. Mukhopadhyay, IET Image Process., 2012, Vol. 6, Iss. 7, pp. 966–975
6. doi: 10.1049/iet-ipr.2011.0472.
7. <https://tatourian.blog/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/>
8. <https://en.wikipedia.org/wiki/Flynn%27s_taxonomy>
9. <http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-computing/>
10. <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#assess-parallelize-optimize-deploy>

**Appendix A**

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

\_\_global\_\_ void matAddKernel(int\* input1,int\* input2,int\* output,int size)

{

int threadX=blockIdx.x\*blockDim.x+threadIdx.x;

int threadY=blockIdx.y\*blockDim.y+threadIdx.y;

if(threadX<size && threadY<size)

{

output[threadY\*size+threadX]=input1[threadY\*size+threadX]+input2[threadY\*size+threadX];

}

}

int main(int argc,char\* argv)

{

printf("An sample CUDA program \n");

/\*Define the input matrix sizes\*/

int matrixSize=100;

/\*Host statements \*/

/\*Allocate memory for matrix hostMatA on host(CPU)\*/

int\* hostInMatA=NULL;

hostInMatA = (int\*)malloc(matrixSize\*matrixSize\*sizeof(int));

if(hostInMatA==NULL)

{

printf("Memory can't be created on host for hostInMat at %d \n",\_\_LINE\_\_);

return -1;

}

/\*Allocate memory for matrix hostMatB on host(CPU)\*/

int\* hostInMatB=NULL;

hostInMatB = (int\*)malloc(matrixSize\*matrixSize\*sizeof(int));

if(hostInMatB==NULL)

{

printf("Memory can't be created on host for hostInMatB at %d \n",\_\_LINE\_\_);

return -1;

}

/\*Allocate memory for matrix hostOutMat on host(CPU)\*/

int\* hostOutMat=NULL;

hostOutMat = (int\*)malloc(matrixSize\*matrixSize\*sizeof(int));

if(hostOutMat==NULL)

{

printf("Memory can't be created on host for hostOutMat at %d \n",\_\_LINE\_\_);

return -1;

}

/\*Allocate memory for matrix hostOutCheck on host(CPU) to check GPU result\*/

int\* hostOutCheck=NULL;

hostOutCheck = (int\*)malloc(matrixSize\*matrixSize\*sizeof(int));

if(hostOutCheck==NULL)

{

printf("Memory can't be created on host for hostOutMat at %d \n",\_\_LINE\_\_);

return -1;

}

/\*Generate input source matrices on host\*/

for(int idx=0;idx<matrixSize\*matrixSize;idx++)

{

hostInMatA[idx]=rand()%100;

hostInMatB[idx]=rand()%150;

}

/\*Device statements\*/

/\* Useful for Error checking for CUDA operations\*/

cudaError\_t err;

/\*Allocate memory on device for input and output matrices \*/

/\*Allocate device memory for devInMatA\*/

int\* devInMatA;

err=cudaMalloc(&devInMatA,matrixSize\*matrixSize\*sizeof(int));

if(err!=cudaSuccess)

{

printf("Device memory can't be allocated for devInmatA %d \n" ,\_\_LINE\_\_);

return -1;

}

/\*Allocate device memory for devInMatB\*/

int\* devInMatB;

err = cudaMalloc(&devInMatB,matrixSize\*matrixSize\*sizeof(int));

if(err!=cudaSuccess)

{

printf("Device memory can't be allocated for devInmatB %d \n" ,\_\_LINE\_\_);

return -1;

}

/\*Allocate device memory for devOutMat\*/

int\* devOutMat=NULL;

err = cudaMalloc(&devOutMat,matrixSize\*matrixSize\*sizeof(int));

if(err!=cudaSuccess)

{

printf("Device memory can't be allocated for devOutmat %d \n" ,\_\_LINE\_\_);

return -1;

}

/\*Copy matrices from host to devices\*/

err = cudaMemcpy(devInMatA,hostInMatA,matrixSize\*matrixSize\*sizeof(int),cudaMemcpyHostToDevice);

if(err!=cudaSuccess)

{

printf("Memory copy failed at %d \n" ,\_\_LINE\_\_);

return -1;

}

err = cudaMemcpy(devInMatB,hostInMatB,matrixSize\*matrixSize\*sizeof(int),cudaMemcpyHostToDevice);

if(err!=cudaSuccess)

{

printf("Memory copy failed at %d \n" ,\_\_LINE\_\_);

return -1;

}

/\*Kernel parameter initialization\*/

/\*Two dimensional Block size initialization \*/

float BlockSizeX=16.0;

float BlockSizeY=16.0;

/\*Two dimensional grid size initialization \*/

int gridSizeX=ceil(float(matrixSize/BlockSizeX));

int gridSizeY=ceil(float(matrixSize/BlockSizeY));

/\*Grid and block initialization\*/

dim3 block(BlockSizeX,BlockSizeY);

dim3 grid(gridSizeX,gridSizeY);

/\*Kernel launch\*/

matAddKernel<<<grid,block>>>(devInMatA,devInMatB,devOutMat,matrixSize);

err = cudaGetLastError();

if(err!=cudaSuccess)

{

printf("Kernel Launch failed at %d \n" ,\_\_LINE\_\_);

return -1;

}

/\*Waiting for the GPU to complete its operation\*/

cudaDeviceSynchronize();

/\*Copy the device result to host \*/

err = cudaMemcpy(hostOutCheck,devOutMat,matrixSize\*matrixSize\*sizeof(int),cudaMemcpyDeviceToHost);

/\*Check the output from GPU against CPU output\*/

for(int idx=0;idx<matrixSize\*matrixSize;idx++)

{

hostOutMat[idx]=hostInMatA[idx]+hostInMatB[idx];

}

bool check=true;

int idx=0;

while(check && (idx<matrixSize\*matrixSize))

{

if(!(hostOutMat[idx]==hostOutCheck[idx]))

{

check=false;

break;

}

idx++;

}

if(check)

printf("Matrix addition completed successfully on GPU");

/\*Free the host memory\*/

if(hostInMatA)

{

free(hostInMatA);

}

if(hostInMatB)

{

free(hostInMatB);

}

if(hostOutMat)

{

free(hostOutMat);

}

if(hostOutCheck)

{

free(hostOutCheck);

}

/\* Free the device memory\*/

if(devInMatA)

{

cudaFree(devInMatA);

}

if(devInMatB)

{

cudaFree(devInMatB);

}

if(devOutMat)

{

cudaFree(devOutMat);

}

}

return 0;

}