CUDA BASED IMPLEMENTATION OF FLAME DETECTION ALGORITHMS IN DAY AND INFRARED CAMERA VIDEOS

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND

ELECTRONICS ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Hasan Hamzaçebi

September 2011

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

_____

Prof. Dr. A. Enis Çetin (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

_____

Assist. Prof. Dr. Sinan Gezici

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

_____

Assoc. Prof. Dr. Ugur Güdükbay

Approved for the Graduate School of Engineering and Science:

_____

Prof. Dr. Levent Onural
Director of Graduate School of Engineering and Science

ABSTRACT

CUDA BASED IMPLEMENTATION OF FLAME DETECTION ALGORITHMS IN DAY AND INFRARED CAMERA VIDEOS

Hasan Hamzaçebi

M.S. in Electrical and Electronics Engineering

Supervisor: Prof. Dr. A. Enis Çetin

September 2011

Automatic fire detection in videos is an important task but it is a challenging problem. Video based high performance fire detection algorithms are important for the detection of forest fires. The usage area of fire detection algorithms can further be extended to the places like state and heritage buildings, in which surveillance cameras are installed. In uncontrolled fires, early detection is crucial to extinguish the fire immediately. However, most of the current fire detection algorithms either suffer from high false alarm rates or low detection rates due to the optimization constraints for real-time performance. This problem is also aggravated by the high computational complexity in large areas, where multi-camera surveillance is required. In this study, our aim is to speed up the existing color video fire detection algorithms by implementing in CUDA, which uses the parallel computational power of Graphics Processing Units (GPU). Our method does not only speed up the existing algorithms but it can also reduce the optimization constraints for real-time performance to increase detection probability without affecting false alarm rates. In addition, we have studied several methods that detect flames in infrared video and proposed an improvement for the

algorithm to decrease the false alarm rate and increase the detection rate of the fire.

ÖZET

GÜNDÜZ VE KIZILÖTESI KAMERA VIDEOLARINDA ALEV TESPIT ALGORITMALARININ CUDA TABANLI GERÇEKLESTIRILMESI

Hasan Hamzaçebi

Elektrik ve Elektronik Mühendisligi Bölümü Yüksek Lisans

Tez Yöneticisi: Prof. Dr. A. Enis Çetin

Eylül 2011

Videoda otomatik ateş tespiti önemli bir görev olup aynı zamanda zorlayıcı bir problemdir. Video tabanlı yüksek performanslı ateş tespit algoritmalarının orman yangını tespitindeki önemi oldukça büyüktür. Ateş tespit algoritmalarının kullanım alanları devlet binaları ve tarihi değeri olan binalar gibi gözetleme kameralarının kurulu olduğu yerleri de içerecek şekilde genişletilebilir. Kontolsüz ortaya çıkan yangınlarda, erken tespit yangının kısa bir sürede söndürülmesine yardımcı olur. Fakat, günümüzde kullanılan çoğu ateş tespit algoritması, gerçek zamanlı video işleyebilmek için yapılan optimizasyonlardan dolayı yüksek yanlış alarm oranlarına veya düşük tespit oranlarına sahiptir. Bu sorun birden fazla kameranın gözetimine ihtiyaç duyulan geniş alanlarda hesaplama karmaşıklığının yükselmesi ile daha da artar. Bu çalışmada amacımız, Grafik İşlemci Ünitesi (GPU)'nin paralel hesaplama gücünü kullanan CUDA'yı kullanarak gündüz kameraları için var olan ateş tespit algoritmalarının hızının arttırılmasıdır. Yöntemimiz sadece var olan algoritmaları hızlandırmak ile kalmaz, bu hızlandırmanın sağladığı faydaları kullanarak gerçek zamanlı işleme

için yapılmış olan optimizasyonlar kaldırılarak, yanlış alarm oranını da etkilemeden, tespit oranını arttırabilir. Buna ek olarak, kızılötesi videolarda alev tespitinde kullanılan bir çok algoritma incelenmiş ve var olan bir algoritmanın yanlış alarm oranını azaltırken tespit oranını arttıran bir yenilik öne sürülmüştür.

Anahtar Kelimeler: Alev Tespiti, Yangın Tespiti, Grafik İşlemci Ünitesi (GPU), Compute Unified Device Architecture (CUDA), Kızılötesi (IR) Videosu, Gündüz Videosu

ACKNOWLEDGMENTS

I would like to express my special thanks to my supervisor Prof. Dr. A. Enis Çetin for his patience, guidance, suggestions and valuable comments throughout this thesis.

I would like to thank to Assist. Prof. Dr. Sinan Gezici and Assoc. Prof. Dr. Uğur Güdükbay for reading this thesis and for being a member of my thesis committee.

I would like to thank Osman Günay, Yusuf Hakan Habiboğlu, Kıvanç Köse and Dr. Murat Gevrekçi for their contributions during the development of this thesis.

I would like to give my special thanks to my friends İsmail Uyanık, Veli Tayfun Kılıç, Serkan Sarıtaş, Oğuz Özcan and Hacı Hasan Coşkun for their support during the development of this thesis.

I would like to thank Bilkent University EE Department and especially to faculty members for giving me this opportunity by teaching me well.

I would like to offer my sincere love to my family, for their support and encouragement in my whole life.

Also, I would like to thank to ASELSAN Inc. for their support and encouragements during my M.S. study.

Contents

List of Figures

List of Tables

Dedicated to my beloved family. . .

Chapter 1

Introduction

In this thesis, the video fire and flame detection (VFD) algorithms are studied and some improvements to the existing algorithms [1, 2] are proposed. VFD algorithms are implemented in Compute Unified Device Architecture (CUDA), which uses the Graphics Processing Unit (GPU).

1.1   Motivation

In recent years, the number of forest fires all around the world is continuously increasing. Accordingly, fire detection in videos has become a popular research topic in the area of signal processing. Current research result in a large number of different fire detection algorithms and techniques for early detection of forest fires.

Heat sensors, smoke sensors and gas detectors are the traditional fire alarm systems. However, these systems have some drawbacks. The most important one is that their usage being only limited to indoors. Moreover, these systems do not provide additional information about the fire such as the size, the direction and

the speed. On the other hand, video based fire detection systems can be used in both indoor and outdoor applications.

Problems of traditional alarm systems are solved with the help of cameras. For instance, Video Fire Detection Systems (VFDS) are used in large public areas like auto-parks, malls and airports. In addition to that VFDS provide faster and reliable detection results. In this thesis, we investigate VFD methods for both ordinary color cameras and infrared cameras.

The existing algorithms in color video fire detection has been successful to preserve natural heritage sites and environment. However, due to the high computational power requirements, algorithms either process low resolution video or produce higher false alarm rates to satisfy real-time processing issues. By using CUDA, which works on GPU, some parts of algorithms can work in parallel to process video faster in a regular computer. This allows the processing of high resolution videos in real-time. Also, the time saved can be used to process additional descriptors with the aim of reducing the false alarm rates and increasing the detection rates.

IR cameras have some advantages over color cameras in flame detection. The first advantage is that ordinary color cameras need sufficient lighting conditions to work properly. Thermal cameras can monitor surrounding areas in low light conditions providing 24 hours surveillance. Lastly, IR cameras provide higher thermal sensitivity but their visual sensitivity is lower than regular cameras. This provides a better realization of the hot objects like flames and also it decreases the effect of external factors such as rain. A major problem with commercially available long wave infrared (LWIR) cameras is that they cannot detect smoke. Obviously, when there is fire there is also smoke.

1.2 Thesis Outline

This thesis is organized as follows. In Chapter 2, we first examine the related works about video flame detection that exist in the literature. We review the GPU architecture and CUDA environment. We also give some examples of our CUDA implementation of a flame detection algorithm and explain the tricks and optimizations that are used to decrease the processing time. Finally, we provide comparison results of our GPU implementation with the Habiboglu's CPU implementation [1] in terms of both detection time and detection rate.

In Chapter 3, we address the infrared video flame detection algorithms. We first study the current infrared video flame detection algorithms. After that, our contributions to the field and the improvements to the current algorithms are proposed. In addition, we examine the GPU implementation and its necessity for these algorithms. Finally, we provide the results of our detection algorithm.

In Chapter 4, we summarize our study and list the possible future works in VFD.

Chapter 2

GPU Implementation of Flame Detection Methods in Videos

Video forest fire detection algorithms require high computational power. With the help of the technological advancements in the last decades, transferring data including high-definition video in real-time is possible. This means that we can collect and process more video data in a given time period. On the other hand, CPU suffers from its limited computational power for these tasks.

CPU technology relies on mostly the core clock speed. Manufacturers increased their clock speed almost 1,000 times in the last 30 years. However, the clock speed cannot be increased further because of the power and heat restrictions of the core and physical limits of the transistor. In order to increase the computational power of the CPU, manufacturers started to produce multi-core processors. Today, even low-end and low-powered processors have also been built as multi-core. Moreover, manufacturers announced their plans to produce processors that have 16-core [3].

Recently, GPUs found application even in supercomputers. Three out of top five supercomputers were built using GPU cores [4]. This is because of the

parallel processing capability of the GPU. Also, the GPU chip manufacturers estimate very large growth in the computing capability (16-fold increase in parallel computing in 3 years [5]) and they continue to increase the performance of processors by putting GPUs and CPUs in the same core like NVIDIA's Tegra and AMD's Fusion. In this way, it reduces the heat dissipated and power used by the processing units. This causes less harm to the environment.

Up to now, we mentioned that GPU is powerful than CPU but we did not mention about its performance when sequential processing is needed. The GPUs have poor quality at sequential processing because they are optimized for parallel processing, thus the GPU and CPU should work in harmony to achieve the best performance. Therefore, the manufacturers try to put CPU and GPU closer in a single chip, which reduces the communication latency.

Data first need to be transferred to the GPU memory for being processed and the processed data should be transferred back to the system memory. The data transfer between memories is costly so it does not worth to process small data on GPU. Therefore, GPU requires large amounts of data to show its computational advantage over the CPU.

Based on all these explanations and given information, it is concluded that there is a need to find the blocks in our algorithm that can be parallelized to harness the power of the GPU and keep the sequential parts in the CPU side not to slow down the GPU.

2.1 Related Work

With the purpose of developing flame detection systems, several methods have been studied to date [6, 7, 8, 9, 10, 11, 12]. Besides them, some of the algorithms on flame detection use different color models such as Gaussian-smoothed histogram [13] or HSI [14] to detect flame pixels. After detecting the flame and non-flame regions they either use temporal variation of flame to make a heuristic flame analysis or use segmentation to find flame colored regions. Piecewise difference between two consecutive frames and segmentation in videos is used to separate flame colored objects from flames. Similarly, some researchers use some background estimation algorithms and Gaussian mixture models to find moving and flame colored pixels [15]. In these algorithms, quasi-periodic behavior in flame boundaries and color variations are detected by using temporal and spatial wavelet analysis.

In a previous work conducted in our lab, Habiboglu designed new flame detection algorithm both in images and videos by using different color models simultaneously [1]. In addition, instead of a pixel by pixel analysis of the whole frame they divided each frame into 16x16 blocks to compensate the computational cost of using different color models and analysis. Figure 2.1 illustrates a sample true detection result and Figure 2.2 illustrates a sample miss detection and false alarm results of this algorithm.
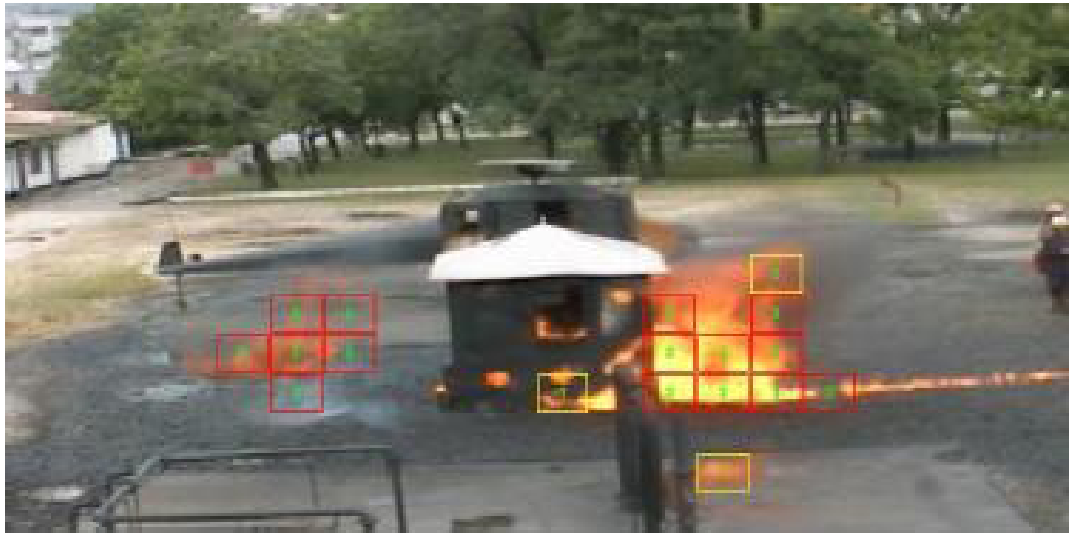
Figure 2.1: True flame detection.

Figure 2.2: Miss detection and false alarm.

2.1.1 Flame Colored Pixel Model

In his study, Chen suggests a flame colored pixel model to classify flame pixels [16]. The flame colored pixel model is used because it is low cost and easy to implement. The analysis of the pixel values of the flame in that domain results in following conditions:

- $R \geq G > B$

- $R > R_T$

where R, G and B are red, green and blue color values of the pixels, respectively, and R_T is a threshold for red color value. The conditions are valid for the flame pixels because the red and yellow colors have fundamental importance in flame regions.

Based on these information we can define flame colored pixels as:

$$\Psi(i, j, n) = \begin{cases} 1 & \text{if } R_{i,j,n} \geq G_{i,j,n} > B_{i,j,n} \text{ and } R_{i,j,n} > 110, \\ 0 & \text{if otherwise.} \end{cases} \tag{2.1}$$

where $R_{i,j,n}$, $G_{i,j,n}$ and $B_{i,j,n}$ are red, green and blue color values of the pixel that is located at position (i, j) of frame n, respectively.

2.1.2 Covariance Matrix Computation

In his work Habiboglu used covariance descriptors [1] and used it to detect flame. Lower-triangle matrix is used because of its symmetry. The covariance matrix formulation that they have used on images is as follows:

$$\hat{\Sigma} = \frac{1}{N-1} \sum_i \sum_j \left(\Phi_{i,j} - \bar{\Phi} \right) \left(\Phi_{i,j} - \bar{\Phi} \right)^T \tag{2.2}$$

where $\Phi_{i,j}$ is a vector containing some parameters of the pixel that is located at position (i,j) of the image or video frame, $\bar{\Phi} = \frac{1}{N} \sum_i \sum_j \Phi_{i,j}$ and $N = \sum_i \sum_j 1$.

As described in Habiboglu [1], we use the following pixel descriptors in $\Phi_{i,j}$ vector:

$$
\begin{aligned}
R_{i,j,n} &= Red(i,j,n) \\
G_{i,j,n} &= Green(i,j,n) \\
B_{i,j,n} &= Blue(i,j,n) \\
I_{i,j,n} &= Intensity(i,j,n) \\
Ix_{i,j,n} &= \left| \frac{\partial I_{i,j,n}}{\partial i} \right| \\
Iy_{i,j,n} &= \left| \frac{\partial I_{i,j,n}}{\partial j} \right| \\
Ixx_{i,j,n} &= \left| \frac{\partial^2 I_{i,j,n}}{\partial i^2} \right| \\
Iyy_{i,j,n} &= \left| \frac{\partial^2 I_{i,j,n}}{\partial j^2} \right| \\
It_{i,j,n} &= \left| \frac{\partial I_{i,j,n}}{\partial n} \right| \\
Itt_{i,j,n} &= \left| \frac{\partial^2 I_{i,j,n}}{\partial n^2} \right|
\end{aligned}
\tag{2.3}
$$

where $Red(i,j,n)$, $Green(i,j,n)$, $Blue(i,j,n)$ and $Intensity(i,j,n)$ are red, green, blue and intensity values of the pixel that located at position (i, j) and frame n, respectively. In other words, the vector $\Phi_{i,j}$ is formed by the above 10 components. First and second derivatives in these formulation are computed over spatial or temporal domain by using [-1 0 1] and [1 -2 1] filters respectively.

The covariance matrix is computed in video, which is captured with FR frames per second. To detect flames they divided video in blocks whose temporal and spatial dimensions are FR and 16×16, respectively. However, to use (2.2), all the video data in the FR frames need to be accumulated to calculate the mean value. To do the calculations as frames arrive, they used another version of the covariance matrix formula that does not need to wait until all the data is collected. Here is the formulation:

$$\hat{\Sigma}(a,b) = \frac{1}{N-1} \left(\sum_i \sum_j \Phi_{i,j}(a)\Phi_{i,j}(b) - \frac{1}{N} \left(\sum_i \sum_j \Phi_{i,j}(a) \right) \left(\sum_i \sum_j \Phi_{i,j}(b) \right) \right) \tag{2.4}$$

Assume $\tau_C = \sum_i \sum_j \sum_n \Psi(i,j,n)$ is the number of flame colored pixels in the spatiotemporal block. $\tau = \sum_i \sum_j \sum_n 1$ is the number of pixels in the spatiotemporal block which has a size of $16 \times 16 \times FR$. If $\tau_C < \frac{3}{5}\tau$ then the block is classified as it has no flame. Otherwise, it is sent to the Support Vector Machine (SVM) [17] classifier to detect flames.

To reduce the computational cost, 10 dimensional $\Phi_{i,j}$ vector is divided into two parts as follows:

$$\Phi_{color}(i,j,n) = \begin{bmatrix} R_{i,j,n} \\ G_{i,j,n} \\ B_{i,j,n} \end{bmatrix} \tag{2.5}$$

$$\Phi_{spatioTemporal}(i,j,n) = \begin{bmatrix} I_{i,j,n} \\ Ix_{i,j,n} \\ Iy_{i,j,n} \\ Ixx_{i,j,n} \\ Iyy_{i,j,n} \\ It_{i,j,n} \\ Itt_{i,j,n} \end{bmatrix} \tag{2.6}$$

and two separate covariance matrices are computed for Φ_{color} and $\Phi_{spatioTemporal}$ by Habiboglu.

Covariance descriptor matrices generated by using $\Phi_{color}(i,j,n)$ and $\Phi_{spatioTemporal}(i,j,n)$ are 3x3 and 7x7, respectively. Since they use the lower triangle or upper triangle parts of the matrices due to their symmetry, all the data in the matrix does not need to be processed. We have to consider 6 and 28 elements in the matrices $\Phi_{color}(i,j,n)$ and $\Phi_{spatioTemporal}(i,j,n)$, respectively. Therefore, we have total of 34 covariance descriptors for the spatiotemporal block of $16 \times 16 \times FR$. By using these 34 descriptors the block is decided as flame or non-flame block.

2.2 GPU Architecture

The first examples of the graphics processors had the ability to render 2D graphics only. The abilities of these processors were developed by adding 3D graphics rendering, pixel shading etc. By using pixel shading and OpenGL or DirectX interface, some scientific calculations can be implemented on GPU. In 2005 NVIDIA announced that they made a chip and programming interface for GPU programming, which enables some complex calculations [3]. They named the cores that can be programmable as CUDA core and the programming language interface as CUDA C. By adding some further features to the design, the GPU can process both graphics and custom calculations. To program the GPU, they added some basic and minimal keywords to the C language to keep it simple and compatible. CUDA C code is compiled by NVIDIA C Compiler (nvcc), which can use GNU Compiler Collection (gcc), Microsoft Visual Studio Compiler (cl) or Intel C++ Compiler (icc).

The basic structures of a CPU and a GPU are presented in Figure 2.3. As it is seen from this figure that GPU uses more transistors in calculation part of the unit compared to the CPU. This enables the GPU to make calculations faster. In addition, it is also seen that GPU has lots of cores inside, which enables the GPU to do more parallel processing compared to the CPU. The GPU has hardware accelerators named as Special Function Unit (SFU) for transcendental functions such as $sin()$, $cos()$ and $log()$ operators and this ability also leads us to make calculations faster.

Furthermore, the automatic scalability property of the GPU is exhibited in Figure 2.4. It is understood that the GPU arranges the jobs that will be processed in parallel with respect to the number of the CUDA cores. This enables the scalability that we do not need to change anything in our programming, when we use different processors that have different number of cores. This feature
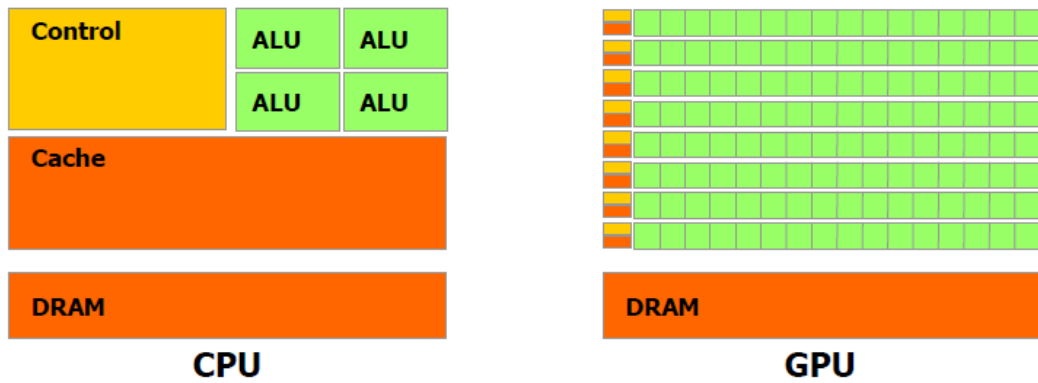
Figure 2.3: Basic structures of CPU and GPU. Here, green units represent Arithmetic Logic Units (ALU).

is only valid for the hardware implementations having same or higher compute capability, which defines the GPU version.
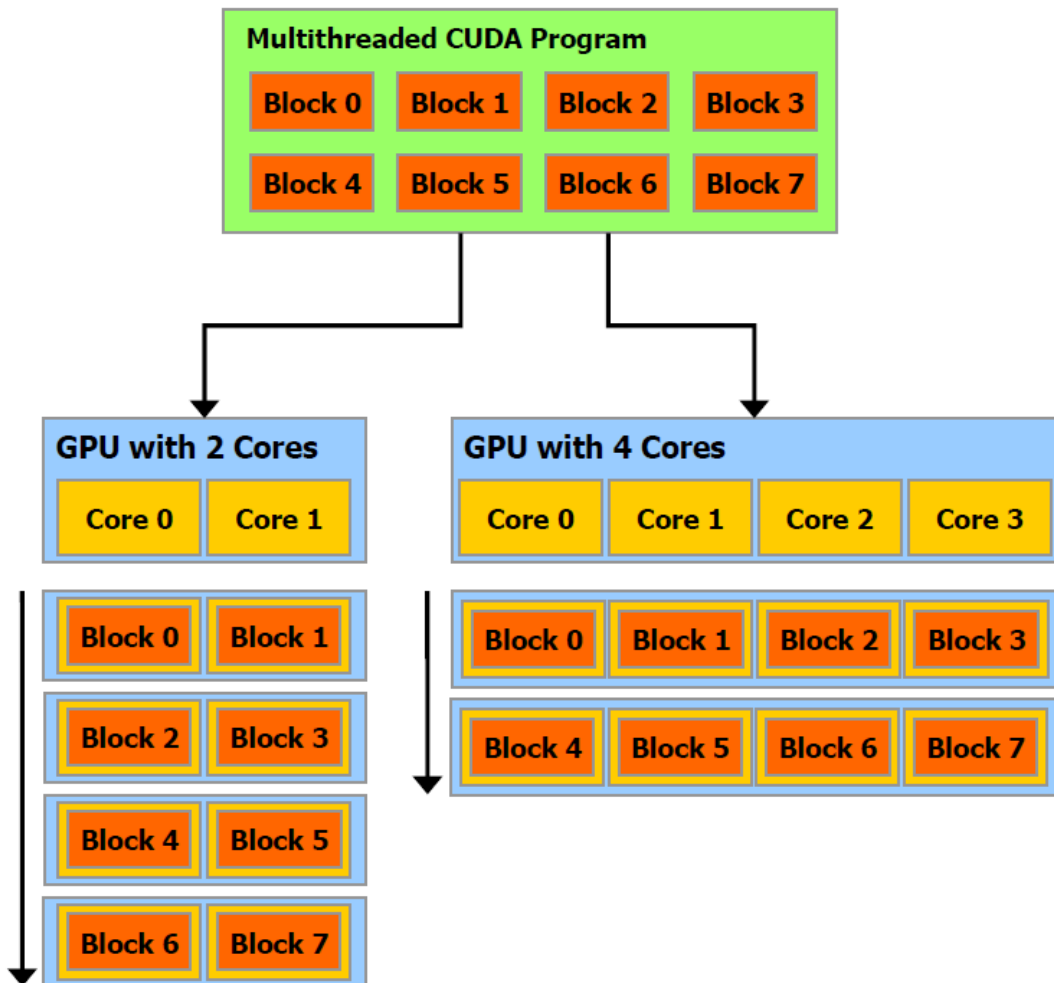
Figure 2.4: Automatic scalability property of the GPU.

Basic job structures that run in parallel are called kernels. Existing GPU designs allow users to run a single kernel at any given time on the GPU. A basic kernel definition can be seen in the Listing 2.1.

Listing 2.1: A basic CUDA kernel structure.

```
1   __global__ void
    AddMat(float **iMat1, float **iMat2, float **oMat, int N, int M)
    {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
6       if (i < N && j < M)
            oMat[i][j] = iMat1[i][j] + iMat2[i][j];
    }


    int main()
11  {
        ...
        dim3 threadsPerBlock(16, 16);
        dim3 numBlocks((N-1)/16+1, (M-1)/16+1);
        AddMat<<<numBlocks, threadsPerBlock>>>(inV1, inV2, outV, N, M);
16  }
```

In this example it is seen that there are some additions to the C language such as <<< ..., ... >>>, __global__, blockIdx, blockDim and threadIdx which are explained below:

- <<< ..., ... >>> is used to call kernel functions. The first parameter is the number of blocks that will run and the second parameter is the number of threads that will run in blocks. Both of them can be three-dimensional (3-D).

- __global__ is a specifier, which implies that the function can be called by only the host code and the code runs on the device. All kernel functions must have it.

- blockIdx is the block number, which is assigned to the block that runs parallel. It can be 3-D and each dimension can be accessed by the properties x, y and z.

- blockDim is the number of threads that runs in a block. It can be 3-D and each dimension can be accessed by the properties x, y and z.

- threadIdx is the thread number, which is assigned to the thread in a block. It can be 3-D and each dimension can be accessed by the properties x, y and z.

The basic structures of grid, block and thread are illustrated in Figure 2.5. It is seen that there is a single grid for a kernel. In a grid there are blocks with a given number up to 3-dimensions and in blocks we have threads that run in parallel. The CUDA cores are optimized for thread context switch so they can handle much more threads than a CPU does. The grid structure that consists of block and thread subunits enables parallel processing across CUDA cores. Threads in a block can communicate with each other via the shared memory of the core. However, a thread cannot communicate with another thread, whose block is different, by using the shared memory. They can communicate with each other by using global memory only. Speed of accessing to the shared memory is same with accessing to the core registers. However, same thing cannot be said for global memory. Therefore, the programmer needs to arrange the threads such that they do the same job and the data they process commonly can be fitted to the shared memory of the core.
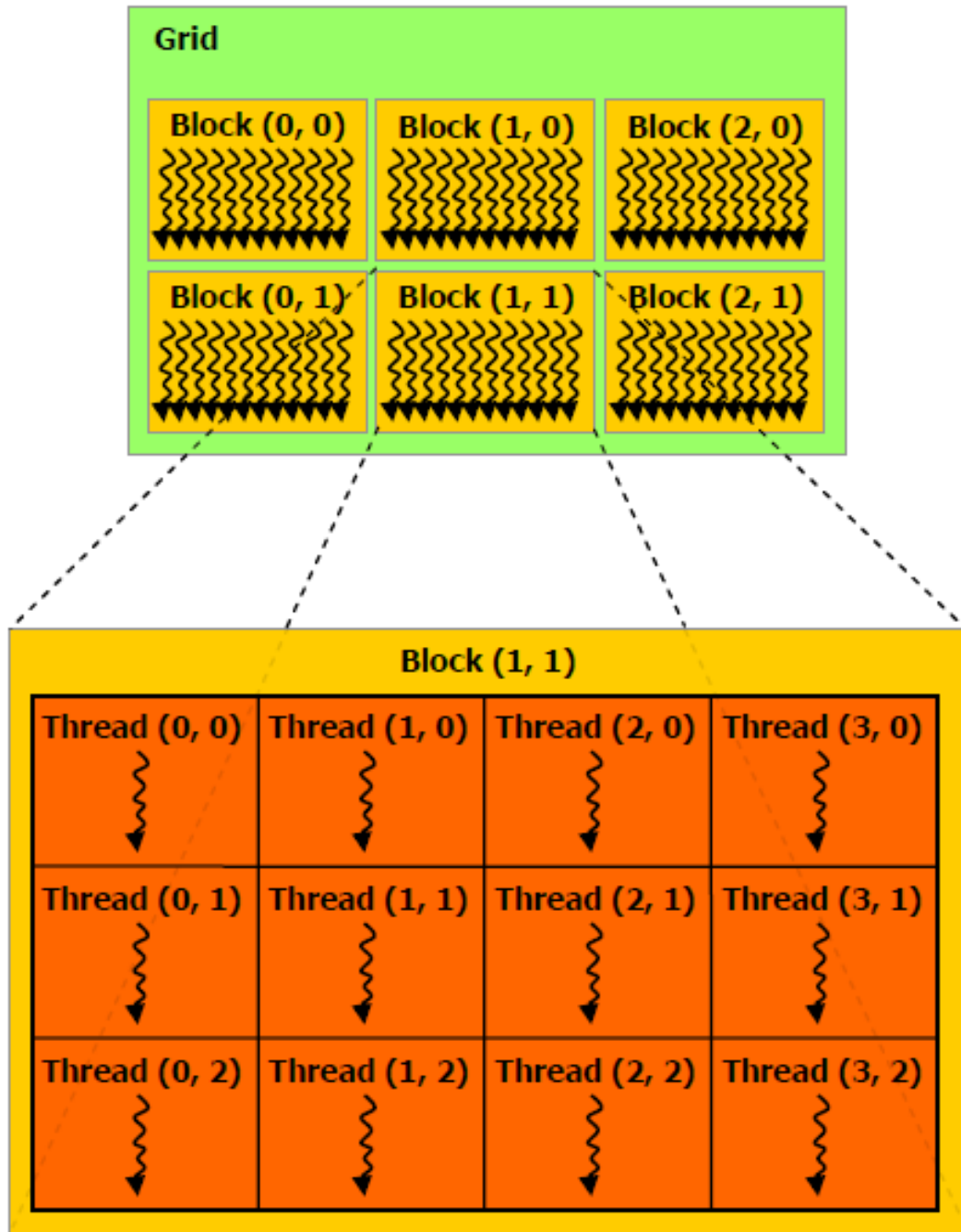
Figure 2.5: Basic structures of grid, block and thread.

In Figure 2.6 what happens when we call a kernel function is described. When we call a kernel function the device starts to execute it and after that it gives the control of the GPU to the CPU. The CPU can then fetch the data and process it or call another kernel.
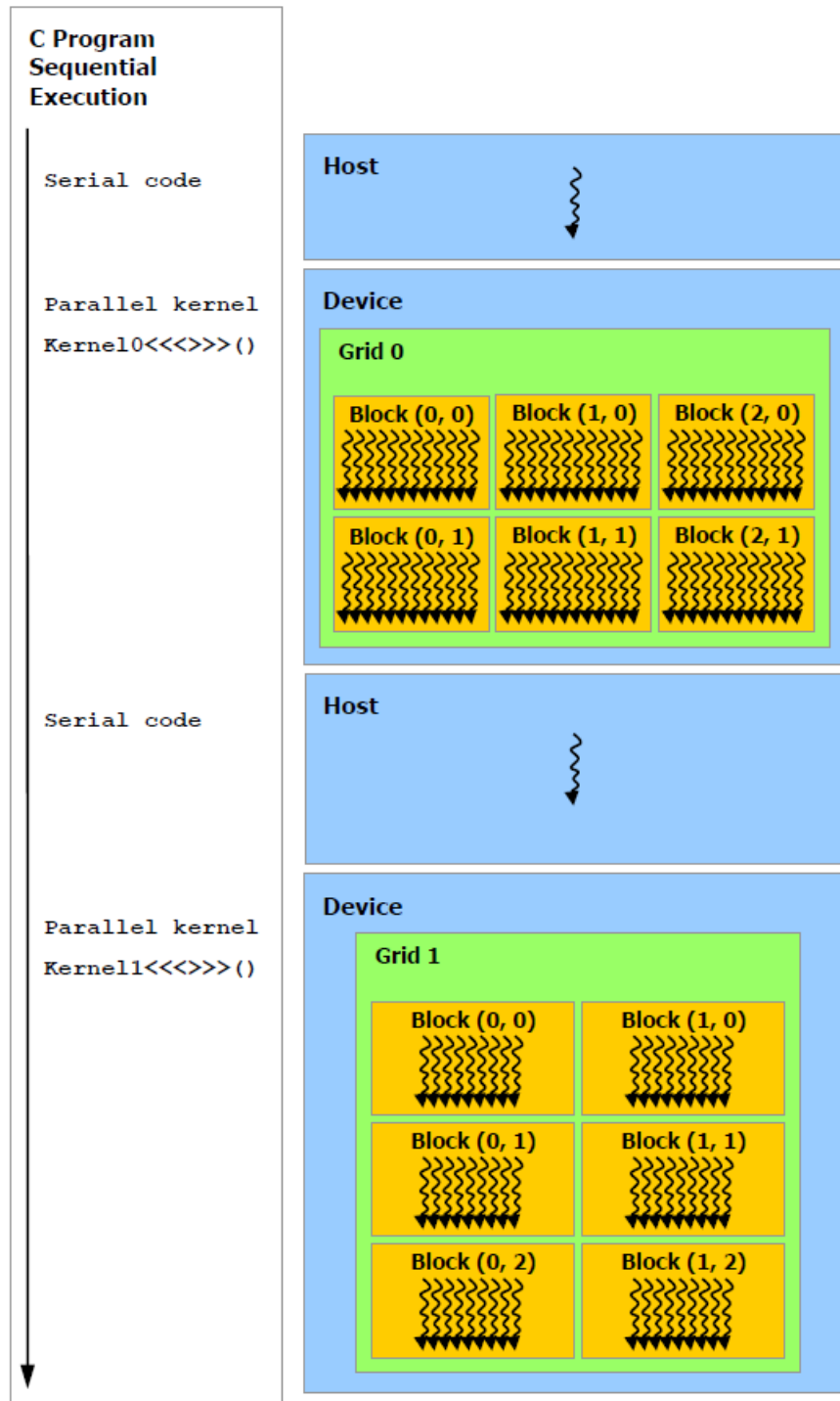
Figure 2.6: Host and device execution sequence.

2.3 Implementation Details of the Flame Detection Algorithm

We use C++ language to transform flame detection algorithms in CPU to our GPU. However, parallel programming on GPU is not straightforward. You need to carefully consider every memory transfer, memory transfer sequence of threads, registers used in one kernel, shared memory used in one block, number of threads per block, number of blocks per grid etc.

Our algorithm consists of the following steps: First the video is decoded from the MJPEG or MPEG formats. Decoded video frame contains raw red, green and blue pixel values in 8-bits unsigned integer representation. From the decoded frame, we calculate the intensity value of every pixel and determine whether the pixel is flame colored or not. After at least three consecutive frames are obtained, we calculate the entries of the covariance matrix. At least three frames are needed because we have a temporal part in our $\Phi_{i,j}$ vectors. We wait until we have $FR/2$ covariance descriptors then sum them up to create the covariance descriptor of our patch block. If we have two sequential patch blocks then we sum them up and create our final descriptors. If in one patch block the number of the flame colored pixels is less than the 60% of all pixels then this block is considered as non-flame region else it is fed to the SVM for final decision[17]. If the SVM decides that there are flame regions then the neighborhood is further examined. If it contains one neighboring flame block then this block has a confidence level of 2. If it contains more than one flame block in the neighborhood then the confidence level of that block is set to 3. The confidence level of the whole frame is the maximum confidence value that the blocks have.

Video decoding is done by the property of the operating system, which is Windows in our case. It is called Video for Windows (VFW). It can decode videos that have the codecs installed and registered on the operating system.

We can select various types of color models as output and we selected raw red, green and blue 8-bit unsigned integer representation.

Calculation of the intensity value of the pixels as

$$I = (299R + 587G + 114B)/1000 \qquad (2.7)$$

and deciding whether the pixels are flame colored or not are done on the GPU. The intensity value is required by spatiotemporal features array, which is represented by $\Phi_{spatioTemporal}(i, j, n)$ in (2.6). Since only features array of the flame colored pixels are used in the calculation of the sum of the covariance matrix, we need to determine which pixel is flame colored and which is not by using flame colored pixel model which is represented by $\Psi(i, j, n)$ in (2.1). There are four versions of the codes used in the calculations and the differences, optimizations done and the results of them are explained in following paragraphs.

First version of the code can be seen in Listing 2.2. In this example, the functions calculate the intensity value and decides whether the pixels are flame colored or not in separate kernels. The functions take the raw color data and pixel count as inputs and give the calculated value as output.

The simple example about calculating the intensity values of pixels and deciding whether the pixels are flame colored or not is shown in Listing 2.2. In this example the raw color data is copied from system memory to GPU memory first. Then the threads per block and number of blocks are determined in accordance with the number of pixels. After that, we have two kernels to be executed. The first one calculates the intensity values and the other decides whether a pixel is flame colored or not. As it can be seen there are two kernels that are executed. Table 2.1 will give the execution times of these two kernels while the number of threads per block change for the first version of the code.

Listing 2.2: Two separate kernels to calculate intensity value and deciding flame colored pixels.

```
__global__ void
IntensityCalc(const unsigned char *rgb, unsigned char *In, int N)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N)
  {
    In[i] = (299*(int)rgb[3*i+2]
            + 587*(int)rgb[3*i+1]
            + 114*(int)rgb[3*i])/1000;
  }
}


__global__ void
isFlameColoredCalc(const unsigned char *rgb, unsigned char *isFC, int N)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N)
  {
    isFC[i] = rgb[3*i+2] >= rgb[3*i+1]
          && rgb[3*i+1] > rgb[3*i]
          && rgb[3*i+2] > 110;
  }
}


void main()
{
  //Copy rgb raw data from host to GPU memory
  cudaMemcpy(rgbHost, rgbGPU, pixelCount, cudaMemcpyHostToDevice);
  //Determine number of threads per block and number of block
  dim3 threadsPerBlock(256);
  dim3 numBlocks( (pixelCount-1)/threadsPerBlock.x+1);
  //Calculate intensity value of the pixels in the frame
  IntensityCalc<<<blocksPerGrid, threadsPerBlock>>>
    (rgbGPU, intensityGPU, pixelCount);
  //Determine whether the pixel is flame colored or not
  isFlameColoredCalc<<<blocksPerGrid, threadsPerBlock>>>
    (rgbGPU, isFlameColoredGPU, pixelCount);
}
```

Table 2.1: Execution time of kernels in Example 1 vs. the number of threads per block.

Number of Pixels	Threads per Block	GPU Time (μs)			Total CPU Time (μs)
		Intensity	Flame Colored	Total	
256000	16	983.588	1007.080	1990.668	1996.778
	32	492.776	507.298	1000.074	1006.073
	64	264.043	272.510	536.553	542.480
	96	189.473	192.242	381.715	387.616
	128	150.297	161.398	311.695	317.412
	256	120.423	118.288	238.711	266.530
	512	124.002	125.287	249.289	255.877
	768	138.597	140.051	278.648	285.042

As seen from the Table 2.1, the CPU works optimally if the threads per block number equals to 256 or 512. To select the threads per block value, the occupancy calculator can be used in CUDA tools. In Table 2.1, GPU Time means the execution time of the code on the GPU. However, the CPU time includes passing the kernel to the GPU and waiting the GPU to complete kernel execution. It can be seen that the differences between total CPU times and total GPU times are around 6μs.

Furthermore, second version of the code can be seen in Listing 2.3. In this example, the function "IntensityAndIsFlameColoredCalc" calculates the intensity value and decides whether the pixels are flame colored or not in a single kernel. The function takes the raw color data and the pixel count as inputs and gives the calculated values as output.

Listing 2.3: Two calculatios in a single kernel.

```
__global__ void
IntensityAndIsFlameColoredCalc(const unsigned char *rgb, unsigned char *In,
    unsigned char *isC, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
    {
        In[i] = (299*(int)rgb[3*i+2]
                + 587*(int)rgb[3*i+1]
                + 114*(int)rgb[3*i])/1000;
        isC[i] = rgb[3*i+2] >= rgb[3*i+1]
            && rgb[3*i+1] > rgb[3*i]
            && rgb[3*i+2] > 110;
    }
}


void main()
{
    //Copy rgb raw data from host to GPU memory
    cudaMemcpy(rgbHost, rgbGPU, pixelCount, cudaMemcpyHostToDevice);
    //Determine number of threads per block and number of blocks
    dim3 threadsPerBlock(256);
    dim3 numBlocks( (pixelCount-1)/threadsPerBlock.x+1);
    //Calculate intensity value of the pixels in the frame
    //and whether the pixels are flame colored or not
    IntensityAndIsFlameColoredCalc<<<blocksPerGrid, threadsPerBlock>>>
        (rgbGPU, intensityGPU, isFlameColoredGPU, pixelCount);
}
```

This example about calculating the intensity values of pixels and deciding whether the pixels are flame colored or not is shown in Listing 2.3. In this example the raw color data is copied from system memory to GPU memory first. Then the threads per block and number of blocks are determined in accordance with the number of pixels. After that, the kernel is executed, which calculates the intensity value and decides whether a pixel is flame colored or not. As it can be seen there is a single kernel that calculates both of the results. The Table 2.2 tabulates the GPU processing time for different numbers of threads per block for a constant number of pixels.

Table 2.2: Execution time of kernel in Example 2 vs. the number of threads per block.

Number of Pixels	Threads per Block	GPU Time (μs)	CPU Time (μs)
	16	1182.440	1185.340
	32	603.177	606.276
	64	332.871	335.734
256000	96	243.583	246.349
	128	193.506	196.400
	256	165.258	168.332
	512	172.048	175.019
	768	190.932	193.914

As seen from Table 2.2, when threads per block is 256 or 512 the GPU works optimally again. It can be seen that we have decreased the calculation time compared to Example 1. Since the amount of computation in each kernel is small and calling a kernel also requires time, we can combine them to increase the performance. It can be seen that the differences between total CPU times

and total GPU times are around 3μs. This is the half of the value observed in Example 1, because we have two kernels in it instead of one.

Third version of the code can be seen in Listing 2.4. Different than Example 2, this code stores the color values in internal registers and uses them from there. Since we use each color data at least twice, we expect that the execution time will decrease. The function takes the raw color data and pixel count as inputs and gives the calculated values as output.

This example about usage of the internal registers is shown in Listing 2.4. In this example, internal registers are used to eliminate latency of retrieving the data from global memory more than once. The Table 2.3 tabulates the GPU processing time for different numbers of threads per block for a constant number of pixels.

Table 2.3: Execution time of kernel in Example 3 vs. the number of threads per block.

Number of Pixels	Threads per Block	GPU Time (μs)
256000	32	510.346
	64	276.558
	96	202.176
	128	157.407
	256	132.868
	512	134.865
	768	147.288

Listing 2.4: Using registers of the CUDA cores.

```cuda
__global__ void
IntensityAndIsFlameColoredCalc2(const unsigned char *rgb, unsigned char *In,
    unsigned char *isC, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
    {
        int r = rgb[3*i+2];
        int g = rgb[3*i+1];
        int b = rgb[3*i];
        In[i] = (299*r + 587*g + 114*b)/1000;
        isC[i] = r >= g && g > b && r > 110;
    }
}


void main()
{
    //Copy rgb raw data from host to GPU memory
    cudaMemcpy(rgbHost, rgbGPU, pixelCount, cudaMemcpyHostToDevice);
    //Determine number of threads per block and number of blocks
    dim3 threadsPerBlock(256);
    dim3 numBlocks( (pixelCount-1)/threadsPerBlock.x+1);
    //Calculate intensity value of the pixels in the frame
    //and whether the pixels are flame colored or not
    IntensityAndIsFlameColoredCalc2<<<blocksPerGrid, threadsPerBlock>>>
        (rgbGPU, intensityGPU, isFlameColoredGPU, pixelCount);
}
```

We can see we have decreased the calculation time compared to Example 2 by looking at Table 2.3. Since threads have very fast access to the internal registers compared to the other memories on the system and we read some data from memory twice, keeping the data to be used closer is a good thing.

Finally the fourth version of the code is presented in Listing 2.5. Before the kernel execution the data is padded with extra data to make it multiples of threads per block. Therefore, this version does not check whether the index is in the limits of the number of pixels or not. We seem to process more data but we expect to gain time because branches taken because of the condition block in the code is eliminated now.

This example which depicts how eliminating condition phrases help us is shown in Listing 2.5. In this example, differently, the "if" condition is removed and data is padded to complete the calculation range. Table 2.4 tabulates the GPU processing time for different numbers of threads per block for a constant number of pixels.

Table 2.4: Execution time of kernel in Example 4 vs. the number of threads per block.

Number of Pixels	Threads per Block	GPU Time (μs)
	32	491.086
	64	262.994
	96	193.776
256000	128	150.574
	256	128.178
	512	128.394
	768	137.974

Listing 2.5: Calling kernel with padded data.

```
__global__ void
IntensityAndIsFlameColoredCalc3(const unsigned char *rgb, unsigned char *In,
3    unsigned char *isC)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int r = rgb[3*i+2];
    int g = rgb[3*i+1];
8    int b = rgb[3*i];
    In[i] = (299*r + 587*g + 114*b)/1000;
    isC[i] = r >= g && g > b && r > 110;
}


13  void main()
{
    //Determine number of threads per block and number of blocks
    dim3 threadsPerBlock(256);
    dim3 numBlocks( (pixelCount−1)/threadsPerBlock.x+1);
18    //Allocate memory with extra space to cover index space of the kernel
    cudaMalloc(&rgbGPU, threadsPerBlock.x * numBlocks.x);
    //Copy rgb raw data from host to GPU memory
    cudaMemcpy(rgbHost, rgbGPU, pixelCount, cudaMemcpyHostToDevice);
    //Calculate intensity value of the pixels in the frame
23    //and whether the pixels are flame colored or not
    IntensityAndIsFlameColoredCalc3<<<blocksPerGrid, threadsPerBlock>>>
        (rgbGPU, intensityGPU, isFlameColoredGPU);
}
```

As seen from Table 2.4, the calculation time is decreased compared to Example 3. Since the condition takes time to execute and in the last block it requires to divide between branches, as expected the time decreases. By using these three optimization techniques we decreased the processing time from 238.711μs to 128.178μs thus our code in examples requires the half of the time to process the data now.

Table 2.5 shows the GPU processing time of the code presented in Listing 2.5 for constant number of threads per block where the number of pixels changes.

Table 2.5: Execution time of kernel in Example 4 vs. the number of pixels.

Number of Pixels	Threads per Block	GPU Time (μs)	Ratio (10^{-4})
256		1.970	76.953
1024		2.133	20.830
10240		5.474	5.346
102400	256	52.841	5.160
256000		128.178	5.007
512000		253.748	4.956
768000		377.944	4.921
10240000		503.577	4.918

In Table 2.5, the GPU time vs. number of pixels is presented. From the data it can be seen that calling a kernel with a small amount of data is not optimum. As the number of data increases the ratio of the processing time to the number of pixels are decreasing. As we have more data to process in one kernel, the optimality increases up to some point. Therefore, if we have low number of data to be processed we can try processing CPU instead of the GPU first. The

CPU can process the data faster because the GPU did not reached its maximum calculation throughput.

In the above examples, we tried to explain the importance of the optimization of the kernel and the importance of the knowledge about the GPU programming. Simple functions like in Example 1 can be optimized like in Example 4. To do this the programmer needs to consider the kernel size and if small kernels exist needs to combine them. If same data will be used more than once in the kernel, the data needs to be put in the internal register. The conditions needed to be escaped as much as possible. If with padding or aligning data the conditions can be eliminated, even if it increases data it can lower the processing time. And finally programmer needs to know the optimum kernel sizes of the compute capability of the device.

After we have at least sequential three frames, we calculate the feature vector $\Phi_{spatioTemporal}(i,j,n)$, $\tau_C = \sum_n \sum_j \sum_j \Psi(i,j,n)$. After we have $FrameRate/2$ frames we calculate $\sum_n \sum_i \sum_j \Phi_{i,j,n}(a)\Phi_{i,j,n}(b)$, $\sum_n \sum_i \sum_j \Phi_{i,j}$ for them. Then we combine the half patch blocks to create final covariance descriptor of our patch block. If in one patch block the number of the flame colored pixels is less than the 60% of all pixels then this block is considered as non-flame region else it is fed to the SVM. If SVM decides that they are flame regions then the neighborhood is examined.

After all patch blocks are classified we give confidence to these patches. The default confidence level of the flame region is 1. If the flame patch has exactly one neighbor that is also a flame patch the confidence level of the flame patch is 2. However if the neighbors that are flame regions are more than 1 then the confidence level is set to 3 indicate the severeness of the detection. After finding the flame regions they are drawn on the video feed to show the place of the flame on the video and give alarm to the user watching the video.

2.4   Results and Summary

We use NVIDIA GeForce GTX 460 as graphics processing unit in our processing time measurements and CUDA Toolkit for harnessing the power of the GPU. In addition, we use AMD Phenom II X2 560 as our CPU for comparison purposes.

We compare the results of our GPU implementation with the Habiboglu's CPU implementation [1]. In our comparisons, we use a total of twelve videos from the dataset of Habiboglu's work, where six of the videos have flame in their frames but the other six do not. The true detection and false alarm rates are calculated in the same way as the Habiboglu's work to be able to have fair comparison results. The definitions of the true detection(T_x) and the false alarm(F_x) rates are given in (2.8) and (2.9), respectively [1]:

$$T_x = \frac{\text{the number of correctly classified frames, which contain flame}}{\text{number of frames which contain flame}} \qquad (2.8)$$

$$F_x = \frac{\text{the number of miss classified frames, which do not contain flame}}{\text{number of frames which do not contain flame}} \qquad (2.9)$$

where the subindex x indicates the confidence level that is used.

Furthermore, the true detection and the false alarm results of the GPU implementation are tabulated in Tables 2.6 and 2.7, respectively.

Table 2.6: True detection rates of the GPU implementation sorted by T_1.

Video Name	T_1	T_2	T_3
posVideo5	2394/2406 (99.5%)	2394/2406 (99.5%)	2394/2406 (99.5%)
posVideo4	1643/1655 (99.3%)	1643/1655 (99.3%)	1643/1655 (99.3%)
posVideo9	651/ 663 (98.2%)	651/ 663 (98.2%)	651/ 663 (98.2%)
posVideo1	281/ 293 (95.9%)	266/ 293 (90.8%)	161/ 293 (54.9%)
posVideo11	166/ 178 (93.3%)	126/ 178 (70.8%)	35/ 178 (19.7%)
posVideo6	225/ 258 (87.2%)	110/ 258 (42.6%)	35/ 258 (13.6%)

Table 2.7: False alarm rates of the GPU implementation sorted by F_1.

Video Name	F_1	F_2	F_3
negVideo3	0/ 160 (0.0%)	0/ 160 (0.0%)	0/ 160 (0.0%)
negVideo8	20/3761 (0.5%)	5/3761 (0.1%)	0/3761 (0.0%)
negVideo7	85/ 541 (15.7%)	0/ 541 (0.0%)	0/ 541 (0.0%)
negVideo5	107/ 439 (24.4%)	45/ 439 (10.3%)	10/ 439 (2.3%)
negVideo6	520/1142 (45.5%)	305/1142 (26.7%)	185/1142 (16.2%)
negVideo4	945/1931 (48.9%)	465/1931 (24.1%)	140/1931 (7.3%)

From Tables 2.6 and 2.7, it can be seen that the true detection and false alarm rates of our GPU implementation are identical with the Habiboglu's results, which shows that we have implemented the algorithm correctly. On the other hand, the processing speeds and processing times of the CPU and GPU implementations for different video resolutions are listed in Tables 2.8 and 2.9, respectively.

Table 2.8: Processing speeds of the GPU and CPU implementations vs. resolution.

Video resolution (px^2)	GPU (fps)	CPU (fps)
320x240	35.00	27.00
640x480	18.25	7.75
960x720	10.00	3.40

Table 2.9: Processing times of the GPU and CPU implementations vs. resolution.

Video resolution (px^2)	GPU (ms)	CPU (ms)	Ratio
320x240	11.90	20.37	1.71
640x480	38.12	112.36	2.94
960x720	83.33	277.45	3.32

Table 2.8 and Table 2.9 demonstrate that the GPU implementation of the algorithm runs faster compared to the CPU implementation. Moreover, it is seen that the ratio of the processing times of the GPU and CPU implementations increase with the video resolution i.e., more than three-fold enhancement in the processing time is reached in high-definition videos.

As a result, this improvement in processing time enables us to process more camera feeds or high definition videos in real-time. Also, because of the time saved it is possible to have additional constraints in the algorithm to increase the detection probability without affecting the false alarm rate.

Chapter 3

Flame Detection Algorithms in IR Videos

Although there is an increasing interest in developing video flame detection algorithms among many researches, the IR flame detection is still not preferred except a few exceptions. However, there is big advantage of working with IR video since it works better in low lighting conditions compared to the color cameras. As another advantage, the thermal perceptibility is higher in the IR spectrum which results in more clear and less disturbed flames. More importantly, hot objects (including the flame itself) obstructed by smoke can be visible in the IR spectrum. This is life-worthy for a fireman to determine the exact location of the flame through smoke. However, in color camera the smoke can block the flame and make it seem invisible to the firemen.

3.1   Related Work

Several methods have been proposed for using IR cameras for automatic flame detection [18, 19, 2, 20]. One of the algorithms uses wavelet transform of the hot

object's contour to extract necessary features [18]. Others use dynamic background subtraction and Otsu's method [21] to detect hot moving objects. Then these objects are used to extract necessary features with the help of some methods whose details will be given in the following paragraphs. In both methods, these features are then used to detect flame in the video.

Toreyin and his colleagues designed and implemented a novel flame detection algorithm based on the wavelet transform for infrared video [18]. Since IR camera sensors measure and display the heat distribution in its field of view, hot objects appear brighter in the IR video compared to the background. Having this in mind they perform the following steps to detect flame in the IR video. First of all, since flame is hot and flickering, the moving bright regions, which are the candidate flame regions, are segmented from the background. The problematic part here is that the images of vehicles, people and animals also appear bright in the IR video, so these objects will be also selected as candidate flame segments during the segmentation process. Fortunately, boundaries of all these objects show very different behaviors such as flame has an irregular boundary which can be easily differentiated from the others. To accomplish this, they first extract the boundaries of these bright regions. To be able to use this boundary information, the centers of mass of the bright objects are calculated as reference points. Then, these reference points are used to compute the distance of the contour from the center in predetermined angles. Additionally, they use wavelet transform to detect irregularities in the boundary. The wavelet transform of this 1-D curve (contour) is calculated and the energy of the high frequency wavelet coefficients are used to classify the contour. In addition to this spatial domain analysis, they also use several temporal analysis techniques to reach a final decision. In these analysis, the information of the flame flickering frequency of around 10 Hz is used. However, due to the aliasing problems, the video needs to be captured at least 20 fps. This temporal information is used for the construction of the Hidden

Markov Model (HMM), which gives the final decision about the classification of the segmented regions about whether they have a flame in them or not.

Bosch et al. [19] propose an object discrimination technique in IR videos. They mentioned that pixel by pixel processing of the frames causes the loss of the geometrical and spatial information and it results in higher false alarm rates. In their work, the process is divided into three parts. First, they get the frames from the video. Next, they extract objects of interest from the frame. Finally, they extract some features from these objects to distinguish between them.

In the first step, to get the images from the video they chose the capturing speed of the frames (measured in fps) according to the event to be classified. As an example, when they consider forest fires they believe one frame per second is enough. However, in vehicle considerations they prefer to have enough frames to study such as frames taken in the interval of a few milliseconds. In processing they only consider the hot regions and the rest is considered as background. The median of the last N images is chosen as background and it is subtracted from the last frame with the aim of making the appearance of the possible objects clear.

In the segmentation part, to reduce the computational requirement some parts of the video (RONI - Region of Not Interest) are ignored. Then they apply the Otsu's thresholding method [21] to find the object regions in the image. After thresholding, they apply morphological opening by using a circle of radius two pixels as the structural element to eliminate the single pixels arise due to noise.

Finally, in the feature extraction part, they calculate the descriptors such as the mean intensity m (3.1), 1-D representation of the boundary and the orientation α (3.6) of the segmented object. The mean intensity is calculated by

$$m = \sum_{i=0}^{L-1} z_i \cdot p(z_i) \tag{3.1}$$

where z_i is the intensity level, $p(z_i)$ is the histogram of the intensity level z_i and L is the number of intensity levels.

The moments for 2-D discrete functions are defined as in (3.2)

$$M_{jk} = \sum_x \sum_y x^j y^k I(x, y) \tag{3.2}$$

where $I(x, y)$ represents the intensity value of the pixel located at (x, y).

By using (3.2), the zeroth and first order moments are calculated in (3.3).

$$
\begin{aligned}
M_{00} &= \sum_x \sum_y I(x, y) \equiv Area \\
M_{10} &= \sum_x \sum_y x I(x, y) \\
M_{01} &= \sum_x \sum_y y I(x, y)
\end{aligned}
\tag{3.3}
$$

Also, from (3.3) the center of mass is calculated.

$$
\begin{aligned}
c_x &= \frac{M_{10}}{M_{00}} \\
c_y &= \frac{M_{01}}{M_{00}}
\end{aligned}
\tag{3.4}
$$

These center of mass points are used to calculate central moment μ_{jk} given by:

$$\mu_{jk} = \sum_x \sum_y (x - c_x)^j (y - c_y)^k I(x, y) \tag{3.5}$$

In [19], Bosch et al. use the center of mass to represent the object boundary as a signature. This signature is calculated by the distance between the boundary and the center of mass for each angle θ. Also, they use μ_{11}, μ_{20} and μ_{02} to calculate the inclination angle α, which is calculated by (3.6).

$$\alpha = \frac{1}{2} arctan \left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}} \right) \tag{3.6}$$

They use these signatures to discriminate between flame regions, people and vehicles.

Verstockt et al. [2, 20] studied the feature-based flame detection by using color and LWIR (long wave infrared) videos. In the infrared part they start with segmenting the moving hot objects and then extracting some features from these objects. For the color video part the process is same except instead of segmenting by moving hot objects, they just look at moving objects. After having features for both IR and color cases, they use theses features to detection of the flame. Since we use infrared video only, we will inspect the LWIR part of the works.

In the moving hot object detection part, a dynamic background subtraction algorithm (3.7), which determines the next background of the scene, is applied. BG_n is the calculated background and I_n is the intensity values of the frame n. If the shift in the $I_n(x, y)$ is bigger than the shift in $BG_n(x, y)$, the pixel is assigned as foreground (FG), otherwise it is labeled as background (BG).

$$BG_{n+1}(x, y) = \begin{cases} \alpha BG_n(x, y) + (1 - \alpha)I_n(x, y), & \text{(x,y) is non-moving} \\ BG_n(x, y) & \text{(x,y) moving.} \end{cases} \quad (3.7)$$

where α is chosen as 0.95 which is because the α determines the update speed of the background, which needs to change little over time.

Later, to avoid noisy objects, a morphological opening with a structural element 3 by 3 square is applied. After the morphological opening, Otsu's method [21] of thresholding is used to distinguish the hot objects by histograms. This method assumes that the image contains two classes of objects. The optimum threshold, which minimizes the intra-class variance (3.8) is calculated iteratively as follows:

$$\sigma^2 = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t) \quad (3.8)$$

37

where ω_i are the probabilities and σ_i^2 are the variances of the classes separated by the threshold t.

After the objects are discriminated, the analysis begins to classify the flame objects. In order to classify three features is proposed: Bounding Box Disorder (BBD), Principal Orientation Disorder (POD) and Histogram Roughness (HR).

Dimensions of the bounding box of the flames varies frequently over short periods of time. Also the variation have a lot of disorder. The BBD is calculated by using the local maxima and minima of the width and the height of the bounding box. The small differences are smoothed by filtering to increase the strength of the feature. The bounding box of the object, which has lots of extrema, will have the BBD value close to 1, whereas less extrema means BBD will be closed to 0. The BBD definition is given as follows:

$$BBD = \frac{|extrema(BB_{width})| + |extrema(BB_{height})|}{N} \qquad (3.9)$$

where N is number of sample points, BB_{width} and BB_{height} are the width and height of the bounding box, respectively.

It is also observed that the disorder in the principle orientation of a flame is higher than the more stationary objects like people, vehicles etc. The principle orientation is calculated by using the ellipse whose major axis has the same second moment with the flame region. The angle α between the major axis of the ellipse and the x-axis is the principle orientation. The POD value is calculated by using the number of maxima and minima as follows:

$$POD = \frac{|extrema(\alpha)|}{N/2} \qquad (3.10)$$

where N is the number of sample points and α is the principle orientations of the region.

Finally, it is observed that the histogram of the flame is rough. The intensity value also varies a lot whereas the other objects localizes on some fixed points and

varies less. The HR (3.11) is calculated by using mean range of the histogram and average disorder of the non-zero bins.

$$HR = \frac{\overline{range(H)}}{N} \times \frac{\overline{|extrema_{bins \neq 0}(H)|}}{N/2} \qquad (3.11)$$

After all three features are calculated, the mean value (3.12) of the features are used to classify the flame from other objects.

$$\frac{BBD + POD + HR}{3} \qquad (3.12)$$

The threshold for the flame is experimentally determined as 0.7. If the mean is over this threshold, the hot object is classified as flame.

3.2 Implementation Details of the IR Flame Detection Algorithm

In this section, we describe the methods that we used to carry out flame detection in IR video. Figures 3.1 and 3.2 contain some IR images that contain flame and other hot objects, respectively.

Figure 3.1: IR image examples that contain flame

Figure 3.2: IR image examples that do not contain flame

To detect flame in IR video we follow the following method which is divided into three parts. First, we used dynamic background subtraction to detect the moving regions, Otsu's method [21] for thresholding the hot objects and morphological opening to eliminate noisy pixels as described in Verstockt et al. [2]. Later we extract some descriptors such as BBD [2], POD [2], Center of Mass Disorder (CMD) and Axes of Bounding Ellipse Disorder (ABED). Finally, the extracted descriptors are used in detection of the flame.

3.2.1 Moving Hot Object Detection

First of all, the moving hot objects are segmented. To detect the moving regions, we used the work developed in Video Surveillance and Monitoring (VSAM) Project at Carnegie Mellon University [22] for background estimation. It is like the one described in the work of Verstockt et al. [2] but the determination of a background pixel is different. To detect the moving pixels (3.13) is used.

$$|I_n(x,y) - I_{n-1}(x,y)| > T_n(x,y) \text{ and } |I_n(x,y) - I_{n-2}(x,y)| > T_n(x,y) \quad (3.13)$$

where $I_n(x,y)$ is the intensity value and $T_n(x,y)$ is the threshold value(3.14) of the pixel located at (x,y) in n^th frame.

The threshold $T_n(x,y)$ is calculated by

$$T_{n+1}(x,y) = \begin{cases} \alpha T_n(x,y) + 5 \times (1-\alpha) \times |I_n(x,y) - B_n(x,y)|, & \text{(x,y) is non-moving} \\ T_n(x,y) & \text{(x,y) moving.} \end{cases}$$
$$(3.14)$$

where α is a constant between zero and one and $B_n(x,y)$ is the estimated background which is calculated in (3.7)

The $T_0(x,y)$ is initialized with some predetermined positive number for all points and $B_0(x,y)$ is initialized with the first image. α is chosen as a constant value close to one.

After the dynamic background subtraction algorithm is applied, we use morphological opening to reduce the effects of the noise as in [2]. The structural element of the morphological operation is chosen as a disk of radius 2.

Figure 3.3 shows an example result of the morphological opening applied on the dynamic background subtraction algorithm.

Figure 3.3: Results of Dynamic Background Subtraction and Morphological Opening using a disk of a radius 2 pixels.

To segment the hot objects, the Otsu's method is used to find a threshold as in [2, 19]. Some IR images that hot object segmentation is applied can be found in Figure 3.4.
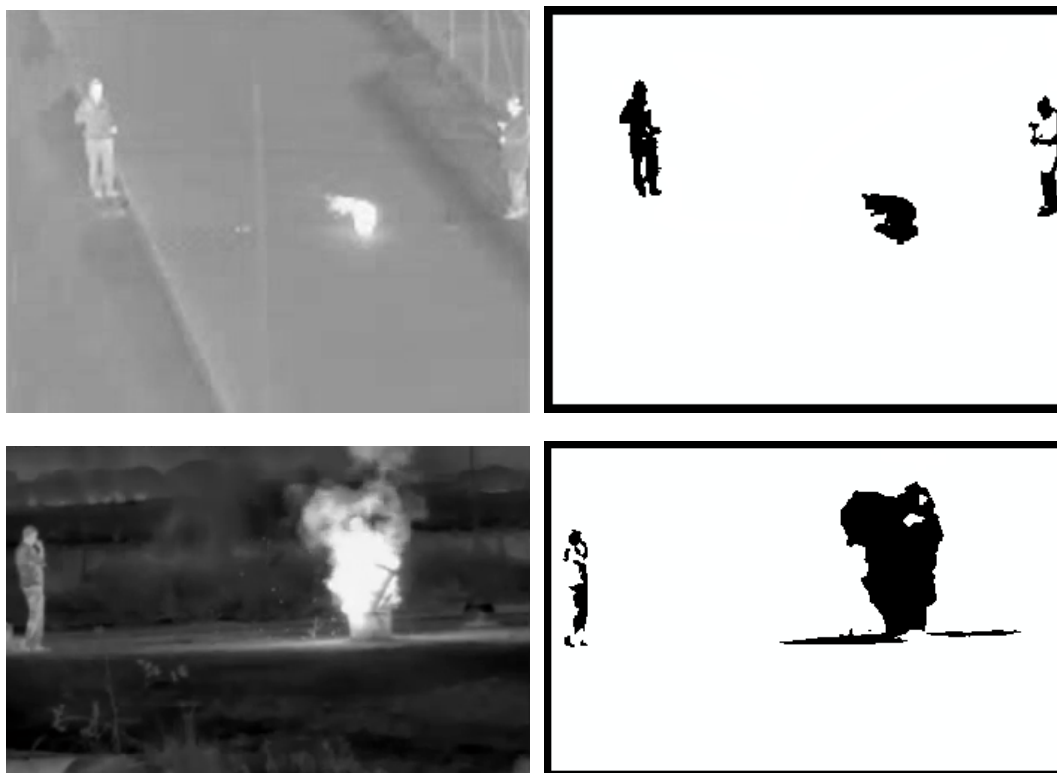

Figure 3.4: Results of Hot Object Segmentation.

3.2.2 Feature Extraction from Flame Regions

We extract four descriptors from the detected hot moving objects. The Bounding Box Disorder and Principle Orientation Disorder are the descriptors used in [2]. The results of extraction of BBD and POD features are in Figures 3.5 and 3.6, respectively.

(a) Flame width

(b) Flame height

(c) Moving person width
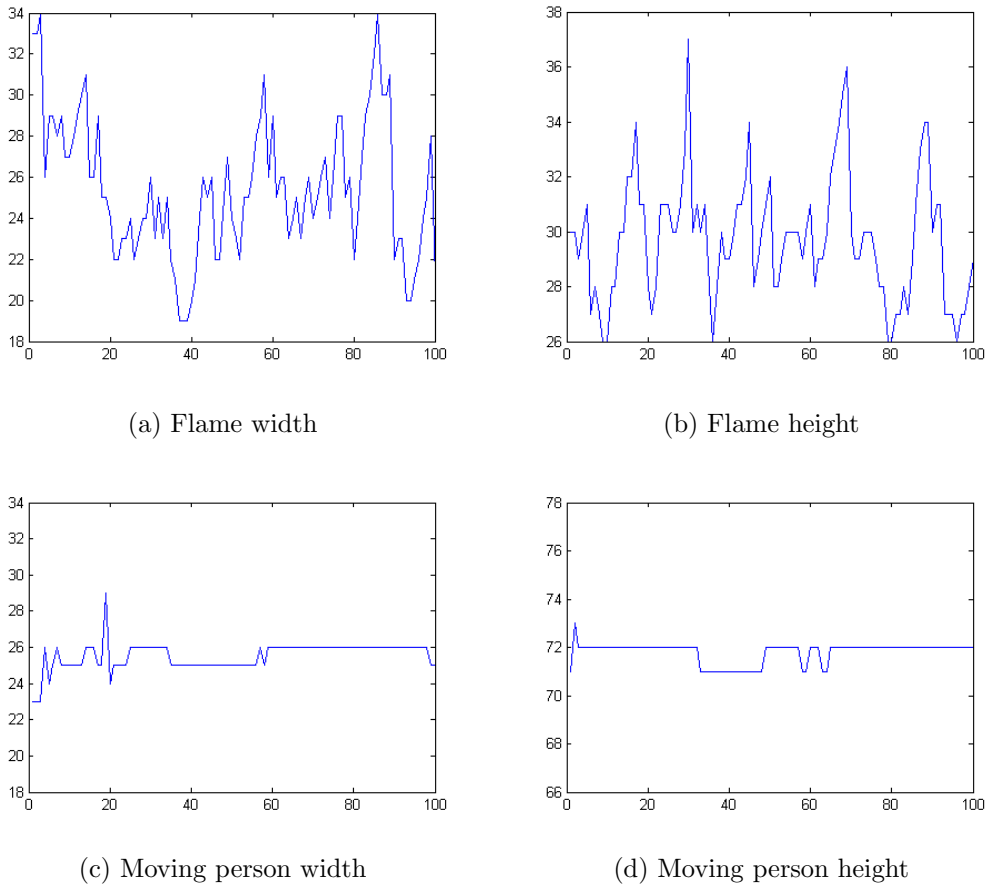
(d) Moving person height

Figure 3.5: Results of Bounding Box Disorder. Vertical and horizontal axes represent lengths (px) and frame numbers, respectively.

As the third descriptor, we use the concept of Center of Mass Disorder (CMD) defined in (3.15). We observed that the disorder in the center of mass of the flame is higher than the other objects such as people and vehicle. The center of mass of the object is calculated by using (3.4). After that, the CMD is calculated
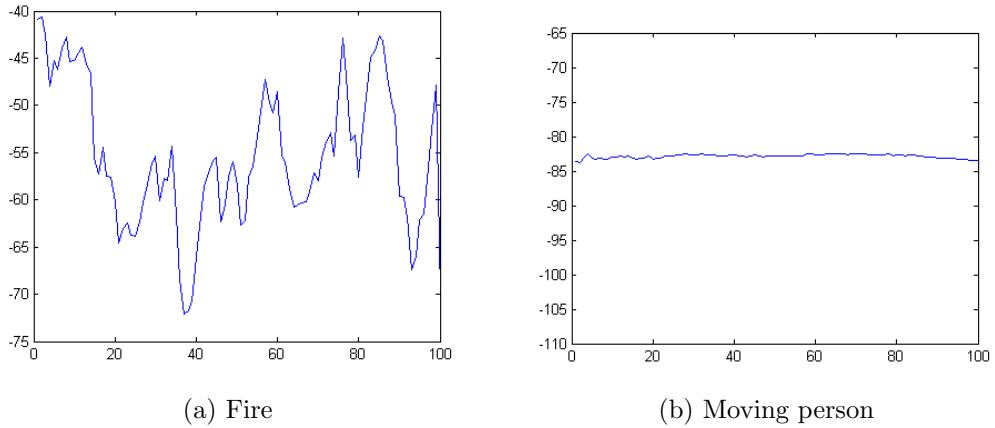
(a) Fire

(b) Moving person

Figure 3.6: Results of Principle Orientation Disorder. Vertical and horizontal axes represent angles (°) and frame numbers, respectively.

by using the extrema points in the data of center of mass points. The data is smoothed before applying the CMD to compensate the effect of noise. The flames have the CMD value close to 1 whereas the more static objects like people and vehicles have the CMD value close to 0. Figure 3.7 shows example data points of this descriptor.

$$CMD = \frac{|extrema(c_x)| + |extrema(c_y)|}{N} \tag{3.15}$$

where N is number of data points and (c_x, c_y) is the position of the center of mass.

The final descriptor is based on the axes of bounding ellipse disorder defined in (3.16). The smallest ellipse is found such that its major axis has the same angle with the principle orientation of the object and it encloses the object. We observed that the disorder in the length of the major and minor axes of this ellipse is higher for flame. The ABED is calculated by using the length of the major and minor axes of this ellipse. Figure 3.8 shows example data points of this descriptor.

44

(a) x coordinates of flame   (b) y coordinates of flame

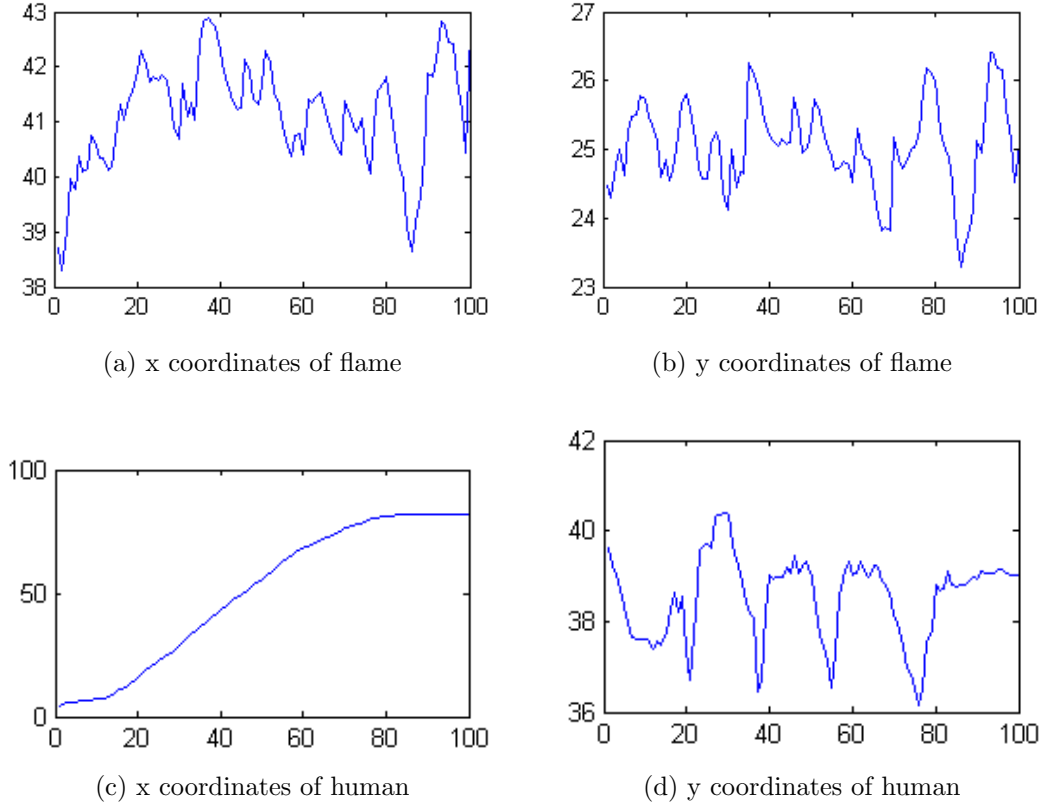(c) x coordinates of human   (d) y coordinates of human

Figure 3.7: Results of Center of Mass Disorder. Vertical and horizontal axes represent positions (px) and frame numbers, respectively.

$$ABED = \frac{|extrema(l_{ma})| + |extrema(l_{mi})|}{N} \tag{3.16}$$

where N is number of data points and l_{ma} and l_{mi} is the length of the major and minor axes of the bounding ellipse described above.

Finally, the extracted descriptors are used in detection of the flame by using the formula in (3.17).

$$FD = \frac{BBD + POD + CMD + ABED}{4} \tag{3.17}$$

where FD stands for flame detection parameter.

45

(a) major-axis for flame   (b) minor-axis for flame

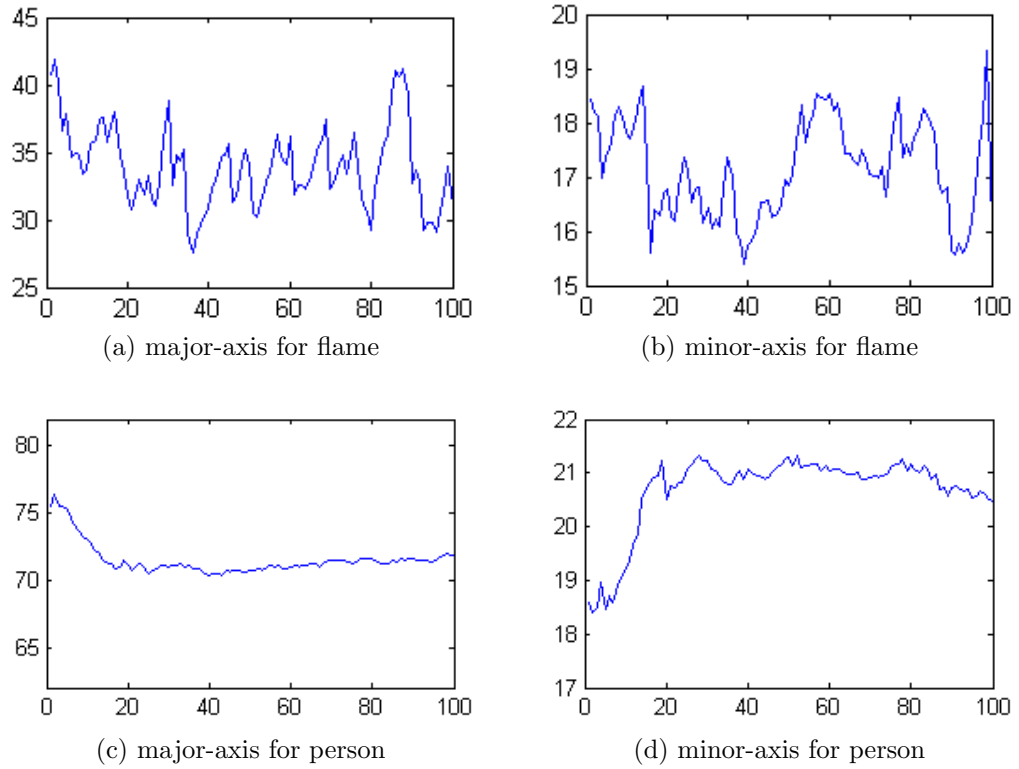(c) major-axis for person   (d) minor-axis for person

Figure 3.8: Results for Axes of Bounding Ellipse Disorder. Vertical and horizontal axes represent lengths (px) and frame numbers, respectively.

In classification, we use thresholding of the descriptors' mean value as similar within [2] for fair comparison. Also, we use SVM [17] for these four descriptors to classify flames.

3.3 CUDA Implementation

As mentioned earlier, the GPU is favorable in speed for the algorithms that can be implemented in a parallel way. Another point when the GPU is advantageous is the case when there are large amount of data to be processed. These are the two cases that make GPU a better environment for most of the algorithms. Unfortunately, our flame detection algorithms for IR videos violate both of these conditions. Therefore, a possible CUDA implementation of this algorithm will bring zero benefit to the CPU implementation.

The very first reason why the CUDA implementation of this algorithm will be slower is the low resolution of our IR videos. Although the morphological opening operation we use is faster in GPU, the overall implementation lasts longer due to the excessive time requirement of the data transfer between the GPU and system memory. When the amount of data to be transferred is large, the transfer time becomes negligible with respect to the long process time of CPU. However, it becomes significantly important for small data since CPU processing becomes shorter. Advantage of using GPU for this algorithm can be observed for high resolution videos which has larger data to be processed.

Secondly, our object detection method consists of simple but many conditional branches which require less computational power. When we look from the CPU perspective, this is a plus since simple key comparisons will take less time. However, when we look from the GPU perspective, this is a big minus since these branches and comparisons (thresholding) brings a sequential and iterative sense to the algorithm. This nature is the reason why the parallel processing is impossible for this algorithm. This is the second important reason why the CUDA implementation of this algorithm will not be advantageous.

3.4 Results and Summary

There are 10 videos in our dataset used throughout this study. Five of these videos contain flame and five of them does not. In both of the algorithms, the flame detection threshold value is chosen as 0.7 as described in the work of Verstock et al. for fair comparison.

Some IR video flame detection examples are illustrated in Figure 3.9. The detected flame regions are shown by a red contour.

(a) posVideo1

(b) posVideo2

(c) negVideo2

(d) negVideo5

Figure 3.9: Some IR image example results of our IR video flame detection algorithm.

Table 3.1: Detection rates of the IR flame detection algorithms

	# of flame	# of detected flame frames		
Video Name	frames	SVM	Thresholding	Verstock et al. [2]
posVideo1	100	100(100%)	100(100%)	100(100%)
posVideo2	305	285(93%)	273(90%)	186(61%)
posVideo3	176	157(89%)	145(82%)	136(77%)
posVideo4	141	138(98%)	136(97%)	115(81%)
posVideo5	279	279(100%)	279(100%)	248(89%)

First three positive videos contain standing people and flame while the others have moving people and flame in them. As seen in Table 3.1, our both SVM and thresholding classifiers outperform the results of Verstock et al. [2].

Table 3.2: False alarm rates of the IR flame detection algorithms

		# detected false flame frames		
Video Name	# of frames	SVM	Thresholding	Verstock et al. [2]
negVideo1	145	0	0	0
negVideo2	241	4	0	3
negVideo3	261	5	3	7
negVideo4	123	0	0	0
negVideo5	111	0	1	3

First negative video contains people leaving a car. The second and third ones contain standing people and moving cars. Finally, the last two have moving cars in them. As seen in Table 3.2, the false alarm rates of our algorithms are lower than the results of Verstock et al. [2]. Because of lack of training data, we could not obtained better SVM results.

As a result of these two tests, we can conclude that our improvements to the algorithm of Verstock et al. [2] increased the detection rate and decreased the false alarm rates.

Chapter 4

Conclusion and Future Work

In this thesis, we investigated the possible CUDA implementations of flame detection algorithms in day and infrared camera videos. Our Graphics Processing Unit (GPU) implementation of an earlier flame detection method in day videos [1] decreases run time of the algorithm as compared to its CPU implementation. During our studies, we observed that the ratio of the processing time of GPU and CPU implementations of the same algorithm increases with video resolution. Our experimental results suggested that an enhancement of more than 3-fold can be achieved in high definition videos. Such a processing is not possible in CPU implementations due to the time constraints, however, a GPU implementation can process even high resolution videos with its parallel processing capability. In addition to these, GPU implementation also enables processing of high definition multi-camera feeds which is not also likely in CPU implementations.

Our results about the GPU implementation of a flame detection algorithm in day video showed that there are significant advantages of using CUDA implementation of these algorithms for flame detection purposes. As a future work, we plan to add additional constrains in the original flame detection algorithm to increase the detection rate while preserving the false alarm rate. This will be

feasible with the CUDA implementation since more comparisons will be possible in a given time interval.

After showing the success of CUDA implementation in day videos, we also investigated the possibility of implementing flame detection algorithms in IR videos. The regular camera based flame detection method cannot be extended to IR flame detection because IR flame regions has almost no texture. In this part, we propose a new flame detection algorithm for IR videos based on [2] by Verstock et al.'s work. Our improvements on this method is to add two new descriptors to consider the disorders in center of mass of bright moving regions and the axis of bounding ellipse of flame regions. These new features increased the detection rate while decreasing the false alarm rate.

Afterwards, we tried to implement our flame detection algorithm for IR videos in GPU. Our goal for the CUDA implementation of this algorithm was to decrease the run time by utilizing the computational abilities of GPU. However, our IR flame detection method has sequential character, which cannot be implemented in a parallel way. Without dividing a task into parallel threads, high transfer time between GPU and system memory dominates the benefits of using GPU. Therefore, CUDA implementation of our flame detection algorithms for IR videos did not speed up the algorithm. As a future work, we plan to develop new flame detection algorithms for IR videos, which are appropriate for parallel processing.

Bibliography

[1] Y. H. Habiboglu, "Fire and flame detection methods in images and videos," Master's thesis, Bilkent University, August 2010.

[2] S. Verstockt, A. Vanoosthuyse, S. Van Hoecke, P. Lambert, and R. Van de Walle, "Multi-sensor fire detection by fusing visual and non-visual flame features," in *4th International Conference on Image and Signal Processing. ICISP'10*, pp. 333–341, 2010.

[3] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to Genearal-Purpose GPU Programming.* Addison-Wesley, October 2010.

[4] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top500 supercomputer sites." `http://www.top500.org/lists/2011/06`, June 2011.

[5] H. Hagedoorn, "NVIDIA Kepler is successor to Fermi." `http://www.guru3d.com/news/nvidia-kepler-is-succesor-to-fermi--due-2011-already/`, September 2010.

[6] B. U. Toreyin, Y. Dedeoglu, U. Gudukbay, and A. E. Cetin, "Computer vision based method for real-time fire and flame detection," *Pattern Recognition Letters*, vol. 27, no. 1, pp. 49–58, 2006.

[7] B. U. Toreyin and A. E. Cetin, "Online detection of fire in video," in *IEEE Conference on Computer Vision and Pattern Recognition. CVPR'07*, pp. 1–5, June 2007.

[8] T.-F. Lu, C.-Y. Peng, W.-B. Horng, and J.-W. Peng, "Flame feature model development and its application to flame detection," in *1st International Conference on Innovative Computing, Information and Control. ICICIC'06*, pp. 158–161, September 2006.

[9] O. Tuzel, F. Porikli, and P. Meer, "Region covariance: A fast descriptor for detection and classification," in *9th European Conference on Computer Vision. ECCV'06*, pp. 589–600, May 2006.

[10] B. U. Toreyin, *Fire detection algorithms using multimodal signal and image analysis.* PhD thesis, Bilkent University, January 2009.

[11] O. Gunay, K. Tasdemir, B. U. Toreyin, and A. E. Cetin, "Video based wild-fire detection at night," *Fire Safety Journal*, vol. 44, pp. 860–868, August 2009.

[12] Y. Dedeoglu, B. U. Toreyin, U. Gudukbay, and A. E. Cetin, "Real-time fire and flame detection in video," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing. ICASSP'05*, pp. 669–672, March 2005.

[13] W. Phillips, III, M. Shah, and N. da Vitoria Lobo, "Flame recognition in video," *Pattern Recognition Letters*, vol. 23(1–3), pp. 319–327, 2002.

[14] W. B. Horng, J. W. Peng, and C. Y. Chen, "A new image-based real-time flame detection method using color analysis," in *Proceedings of IEEE International Conference on Networking, Sensing and Control. ICNSC'05*, pp. 100–105, March 2005.

[15] B. U. Toreyin, Y. Dedeoglu, and A. E. Cetin, "Flame detection in video using hidden Markov models," in *IEEE International Conference on Image Processing. ICIP '05*, vol. 2, pp. 1230–1233, September 2005.

[16] T.-H. Chen, P.-H. Wu, and Y.-C. Chiou, "An early fire-detection method based on image processing," in *International Conference on Image Processing. ICIP '04*, vol. 3, pp. 1707–1710, October 2004.

[17] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 1–27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[18] B. U. Toreyin, R. G. Cinbis, Y. Dedeoglu, and A. E. Cetin, "Fire detection in infrared video using wavelet analysis," *Optical Engineering*, vol. 46, no. 6, 067204, 9 pages, 2007.

[19] I. Bosch, S. Gomez, R. Molina, and R. Miralles, "Object discrimination by infrared image processing," in *Bioinspired Applications in Artificial and Natural Computation*, vol. 5602 of *Lecture Notes in Computer Science*, pp. 30–40, Springer Berlin / Heidelberg, 2009.

[20] S. Verstockt, S. Van Hoecke, N. Tilley, B. Merci, B. Sette, P. Lambert, C. Hollemeersch, and R. Van de Walle, *Hot topics in video fire surveillance*, pp. 443–458. Video Surveillance, Intech, 2011.

[21] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 9, pp. 62–66, Jan 1979.

[22] R. Collins, A. Lipton, T. Kanade, H. Fujiyoshi, D. Duggins, Y. Tsin, D. Tolliver, N. Enomoto, and O. Hasegawa, "A system for video surveillance and monitoring," Tech. Rep. CMU-RI-TR-00-12, Robotics Institute, Carnegie Mellon University, May 2000.